

Technical Reference Guide

- [Quick Reference](#)
 - [RDML Commands](#)
 - [RDMLX Commands](#)
 - [Built-In Functions by Category](#)
 - [System Variables](#)
 - [Formats, Values and Codes](#)
- [Fields](#)
- [Rules and Triggers](#)
- [Files](#)
- [Components](#)
- [Weblets](#)
- [Processes and Functions](#)
- [RDML Commands](#)
- [RDMLX Commands and Full RDMLX](#)
- [Built-In Functions](#)
- [Intrinsics](#)
- [System and Multilingual Variables](#)
- [Formats, Values and Codes](#)
- [Common RDML/X Parameters & BIF Notes](#)
- [Template Commands and Variables](#)
- [External Resources](#)
- [Windows and Linux Considerations](#)
- [Execution Control](#)
- [Error Messages](#)
- [Appendix A. Other Vendor Built-In Functions](#)

Edition Date July 22, 2015

© LANSA

Quick Reference

[RDML Commands](#)

[RDMLX Commands](#)

[Built-In Functions by Category](#)

[System Variables](#)

[Formats, Values and Codes](#)

RDML Commands

Refer to the Visual LANSA Developer Guide for:

- [Sample RDML Programs](#) and
- [Producing Reports Using LANSA](#).

The following commands make up the complete LANSA RDML programming language.

The hyperlink from the command column will take you to the Command's description and syntax diagram.

The [✓](#) in the example column will take you straight to the Command's example.

* indicates that there may be portability conflicts. Check the command for details.

Command	Description	Example	*
ABORT	Abort function with error message	✓	
ADD_ENTRY	Add new entry to a list	✓	
BEGIN_LOOP	Begin a processing group	✓	
BEGINCHECK	Begin validation check block	✓	
BROWSE	Browse and select records from a file	✓	*
CALL	Call a program or process/function	✓	*
CALLCHECK	Validate data by calling a program	✓	*
CASE	Begin a case condition	✓	
CHANGE	Change content of field(s)	✓	*
CHECK_FOR	Check for record in a file	✓	*
CLOSE	Close file(s)	✓	
CLR_LIST	Clear a list	✓	
COMMIT	Commit database changes	✓	*
CONDCHECK	Validate data by checking a condition	✓	
CONTINUE	Continue next iteration of loop	✓	

DATECHECK	Validate data as a valid date or in date range	<u>✓</u>	
DEF_ARRAY	Define an array	<u>✓</u>	
DEF_BREAK	Define report break line(s)	<u>✓</u>	*
DEF_COND	Define a condition	<u>✓</u>	
DEF_FOOT	Define report foot line(s)	<u>✓</u>	*
DEF_HEAD	Define report heading line(s)	<u>✓</u>	*
DEF_LINE	Define report detail line(s)	<u>✓</u>	*
DEF_LIST	Define a browse or working list	<u>✓</u>	*
DEF_REPORT	Define report attributes	<u>✓</u>	
DEFINE	Define a field in this function	<u>✓</u>	
DELETE	Delete record(s) from a file	<u>✓</u>	*
DISPLAY	Display information on a workstation	<u>✓</u>	*
DLT_ENTRY	Delete an entry from a working list	<u>✓</u>	
DLT_LIST	Delete a list	<u>✓</u>	
DOUNTIL	Do until a condition is true.	<u>✓</u>	
DOWHILE	Do while a condition is true.	<u>✓</u>	
ELSE	Else, if an IF condition is not true	<u>✓</u>	
END_LOOP	End a processing loop	<u>✓</u>	
ENDCASE	End a case condition	<u>✓</u>	
ENDCHECK	End a validation check block	<u>✓</u>	
ENDIF	End an IF condition	<u>✓</u>	
ENDPRINT	End all printing	<u>✓</u>	*
ENDROUTINE	End a subroutine	<u>✓</u>	*
ENDSELECT	End a select loop	<u>✓</u>	
ENDUNTIL	End a DOUNTIL loop	<u>✓</u>	

ENDWHILE	End a DOWHILE loop	✓	
EXCHANGE	Exchange information with another function	✓	
EXEC_CPF	Execute a CPF command	Discontinued	*
EXEC_OS400	Execute an IBM i operating system	✓	*
EXECUTE	Execute a subroutine	✓	
EXIT	Exit from LANSAs	✓	
FETCH	Fetch a record from a file	✓	*
FILECHECK	Validate data by checking against a file	✓	*
FUNCTION	Define function control options	✓	*
GET_ENTRY	Get entry from a list	✓	
GOTO	Pass control to another command	✓	
GROUP_BY	Group fields under a common name	✓	
IF	If a condition is true	✓	
IF_ERROR	If a validation error was detected	✓	
IF_KEY	If a key was used at workstation	✓	*
IF_MODE	If screen display mode is	✓	*
IF_NULL	If field(s) are null	✓	
IF_STATUS	If I/O status flag is	✓	
INCLUDE	Include RDML from another function	✓	*
INSERT	Insert a new record into a file	✓	*
INZ_LIST	Initialize a list	✓	
KEEP_AVG	Keep average of fields	✓	
KEEP_COUNT	Keep count of fields	✓	
KEEP_MAX	Keep maximum of fields	✓	
KEEP_MIN	Keep minimum of fields	✓	

KEEP_TOTAL	Keep total of fields	<u>✓</u>	
LEAVE	Leave current loop	<u>✓</u>	
LOC_ENTRY	Locate an entry in a list	<u>✓</u>	
MENU	Transfer to process main menu	<u>✓</u>	*
MESSAGE	Issue a message	<u>✓</u>	*
ON_ERROR	On a validation error being detected	<u>✓</u>	
OPEN	Open file(s)	<u>✓</u>	*
OTHERWISE	If no WHEN commands are satisfied	<u>✓</u>	
OVERRIDE	Override field's dictionary attributes	<u>✓</u>	
POINT	Point file to another file/library/member	<u>✓</u>	*
POP_UP	Present a pop up window on a workstation	<u>✓</u>	*
PRINT	Print line(s) on a report	<u>✓</u>	
RANGECHECK	Validate data by a range of values check	<u>✓</u>	
RENAME	Rename a field within a file	<u>✓</u>	*
REQUEST	Request information from the workstation	<u>✓</u>	*
RETURN	Return from a subroutine	<u>✓</u>	
ROLLBACK	Roll back database changes	<u>✓</u>	*
SELECT	Select records from a file	<u>✓</u>	*
SELECTLIST	Select entries from a list	<u>✓</u>	
SELECT_SQL	Select records from a file using SQL	<u>✓</u>	*
SET_ERROR	Set a validation error	<u>✓</u>	
SET_MODE	Set screen display mode	<u>✓</u>	*
SKIP	Skip to line "n" on a report	<u>✓</u>	
SORT_LIST	Sort list into a nominated order	<u>✓</u>	*

SPACE	Space "n" lines on a report	<u>✓</u>	
SUBMIT	Submit a program or process to batch	<u>✓</u>	*
SUBROUTINE	Define a subroutine	<u>✓</u>	*
SUBSTRING	Substring one field into another field	<u>✓</u>	
TRANSFER	Transfer control to another function	<u>✓</u>	*
UPD_ENTRY	Update an entry in a list	<u>✓</u>	
UPDATE	Update record(s) in a file	<u>✓</u>	*
UPRINT	Print information (unformatted)	<u>✓</u>	*
USE	Use a Built In Function	<u>✓</u>	
VALUECHECK	Validate data by a list of values check	<u>✓</u>	
WHEN	When condition is true (within CASE)	<u>✓</u>	

RDMLX Commands

ASSIGN	ASSIGN command can be specified without the command name to assign a value to one or more variables
ATTRIBUTE	Enables the assignment of declarative attributes to the features of a component class.
BEGIN_COM	Starts a component definition.
DEFINE_COM	Defines a component inside another.
DEFINE_EVT	Creates a custom-specified event.
DEFINE_MAP	Creates input and output parameters.
DEFINE_PTY	Creates a custom-specified property.
END_COM	Ends a component definition.
ENDFOR	Ends a FOR loop.
ENDROUTINE	Ends EVTRoutine, MTHROUTINE & PTYROUTINE block.
EVTRoutine	Defines a routine for an event.
FOR	Enables the definition of a looping block of code.
IF_REF	Compares references of component variables.
IMPORT	Used to include function libraries into an object.
INVOKE	Invokes a method.
MTHROUTINE	Creates a custom-specified method.
PERFORM	Enables the calling of a component method, library function or intrinsic feature.
PTYROUTINE	Creates a custom-specified property.
SELECT_SQL Free Format	Select records using SQL that is valid for any database engine.
SET	Sets a property.

SET_REF Creates component reference.

SIGNAL Triggers a custom-specified event.

WEB_MAP Defines inputs and outputs of WEBROUTINE

WEBROUTINE Defines routine to process an input and/or output request of a Web Application Module (WAM)

Built-In Functions by Category

The platform applicable for each BIF and a description has been presented in each category table.

Application Execution Control Built-In Functions	List Handling Built-In Functions
Authority Built-In Functions	Locking Built-In Functions
Client/Server Support Built-In Functions	Mathematical Built-In Functions
Data Area Built-In Functions	Messages and Message Handling Built-In Functions
Data Queue Built-In Functions	Message Box Built-in Functions
Date and Time Built-In Functions	Miscellaneous Built-In Functions
Domino Integration Built-In Functions	Other Vendor Built-In Functions
Email Handling Built-In Functions	Process Related Built-In Functions
Enhanced 5250 User Interface Built-In Functions	Rule/Trigger Related Built-In Functions
Exchange List Built-In Functions	Space Built-In Functions
Export/Import/Deployment Built-In Functions	Spool File Built-In Functions
Field and Component Related Built-In Functions	String Handling Built-In Functions
File Related Built-In Functions	Template Related Built-In Functions
Function Related Built-In Functions	Unique Operating System and Platform Access Built-In Functions
LANSA Composer Built-In Functions	Web Built-In Functions
LANSA Integrator Built-In Functions	Zip Built-In Functions

Application Execution Control Built-In Functions

Built-In Function	Description
GET_SESSION_VALUE	Returns the value for a specified X_RUN parameter.
SET_FOR_HEAVY_USAGE	Set for heavy usage mode.
SET_FOR_LIGHT_USAGE	Set for light usage mode.
SET_SESSION_VALUE	Set or reset a Visual LANSa session value.

Authority Built-In Functions

Built-In Function	Description
CHECK_AUTHORITY	Check authority on object.
GET_AUTHORITIES	Retrieves a list of authorities and returns it in a variable length working list.
SET_AUTHORITY	Sets user authority to a LANSAs object.

Client/Server Support Built-In Functions

Built-In Function	Description
CALL_SERVER_FUNCTION	Calls a function on a server.
CHANGE_IBMI_SIGNON	Changes the password of the user profile on the IBM i server.
CHECK_IBMI_SIGNON	Checks the status of the user profile on the IBM i server.
CONNECT_FILE	Connects a file to a server. See Database Connection Notes
CONNECT_SERVER	Connects to a server. See Database Connection Notes
DEFINE_ANY_SERVER	Defines details of a LANSAsystem to be used as a server.
DEFINE_DB_SERVER	Defines database for redirected files. See Database Connection Notes
DEFINE_OS_400_SERVER	Defines an IBM i server machine.
DEFINE_OTHER_SERVER	Defines a non-IBM i server system.
DISCONNECT_FILE	Disconnects a file from a server.
DISCONNECT_SERVER	Disconnects from a server.

Data Area Built-In Functions

Built-In Function	Description
--------------------------	--------------------

GET_CHAR_AREA	Get a character value from a data area.
-------------------------------	---

GET_NUM_AREA	Get a numeric value from a data area.
------------------------------	---------------------------------------

PUT_CHAR_AREA	Put a character value into a data area.
-------------------------------	---

PUT_NUM_AREA	Put a numeric value into a data area.
------------------------------	---------------------------------------

Data Queue Built-In Functions

Built-In Function	Description
RCV_FROM_DATA_QUEUE	Receive working list entry(s) from a data queue
SND_TO_DATA_QUEUE	Send working list entry(s) to a data queue

Date and Time Built-In Functions

- [Date Formats](#)

Built-In Function	Description
CONVERTDATE	Converts format of alphanumeric date.
CONVERTDATE_NUMERIC	Converts format of numeric date
DATEDIFFERENCE	Calculates the difference between two dates.
DATEDIFFERENCE_ALPHA	Calculates the difference between two alphanumeric dates
FINDDATE	Finds date "n" days after/before a given date.
FINDDATE_ALPHA	Finds date "n" days after/before a given alphanumeric date

Domino Integration Built-In Functions

Built-In Function	Description
DOM_ADD_FIELD	Adds a field to an open data note using the item name.
DOM_ADD_ITEM	Adds an item to an open data note using the item name.
DOM_CLOSE_DATABASE	Closes an opened Domino/Notes Database on a local or remote Domino server.
DOM_CLOSE_FILE	Closes an opened Domino/Notes File on a local or remote Domino server.
DOM_CLOSE_DOCUMENT	Closes an open document when no longer required.
DOM_CREATE_DOCUMENT	Creates a new document/data note in memory within an opened database.
DOM_DELETE_DOCUMENT	Deletes a document/data note from the database using the given Note ID.
DOM_DELETE_FIELD	Deletes a field from an open document/data note using the item name.
DOM_DELETE_ITEM	Deletes an item from an open document/data note using the item name.
DOM_END_SEARCH_DOCS	Releases all memory that was allocated for search process.
DOM_EXECUTE_AGENT	Executes an Agent.
DOM_GET_FIELD	Gets a field from an open data note using the item name.
DOM_GET_ITEM	Gets an item from an open data note using the item name.
DOM_GET_NXT_DOCUMENT	Retrieves Note ID from the ID Table.

DOM_CLOSE_DATABASE	Closes a Domino/Notes Database on a local or remote Domino server.
DOM_CLOSE_FILE	Closes a Domino/Notes File on a local or remote Domino server.
DOM_OPEN_DATABASE	Opens a Domino/Notes Database on a local or remote Domino server
DOM_OPEN_DOCUMENT	Opens the specified document within a database using the given Note ID.
DOM_SEARCH_DOCUMENTS	Searches a database for documents/data notes matching selection criteria or using a previously created view.
DOM_UPDATE_DOCUMENT	Updates a document/data note in the database.
DOM_UPDATE_FIELD	Updates an existing field to an open data note using the item name.
DOM_UPDATE_ITEM	Updates an existing item to an open data note using the item name.

Email Handling Built-In Functions

Also see

[Email Built-In Functions Notes](#)

Built-In Function	Description
MAIL_START	Initialize a mail message
MAIL_ADD_TEXT	Append message text
MAIL_ADD_ATTACHMENT	Add attachment file
MAIL_ADD_RECIPIENT	Add a recipient
MAIL_ADD_ORIGINATOR	Add an originator
MAIL_SET_SUBJECT	Set the mail message subject
MAIL_SET_OPTION	Set O/S specific mail option
MAIL_SEND	Send mail message

Enhanced 5250 User Interface Built-In Functions

Built-In Function	Description
ADD_DD_VALUES	Add dropdown values.
ALLOW_EXTRA_USER_KEY	Allow an extra user function keys to be used.
BUILD_WORK_OPTIONS	Build option lists for a "Work With" Driver
DROP_DD_VALUES	Drop dropdown values.
DROP_EXTRA_USER_KEYS	Drop all extra user function keys assigned.
SET_ACTION_BAR	Make pull down choices available / unavailable.
SET_DD_ATTRIBUTES	Set attributes of a drop down field.
SHOW_HELP	Modally display help

Exchange List Built-In Functions

Built-In Function	Description
EXCHANGE_ALPHA_VAR	Put an alphanumeric value on the exchange list.
EXCHANGE_NUMERIC_VAR	Put a numeric value on the exchange list.

Export/Import/Deployment Built-In Functions

Built-In Function	Description
EXPORT_OBJECTS	Creates LANSAs Import formatted files LANSAs objects specified.
GET_BIF_LIST	Searches for BIF name and returns BIF details.
GET_COMPOSITION	Returns the list of objects that comprise a LANSAs Object.
GET_ENVIRONMENTS	Return a list of the Environment Names and a build environment indicator.
GET_LICENSE_STATUS	Retrieve the status of LANSAs licenses in this LANSAs system as at specified particular date
GET_PROPERTIES	Returns the version details for a single LANSAs object.
GET_TASK_DETAILS	Retrieves a list of all objects modified under the specified task.
GET_TASK_LIST	Reads LX_F75 to return a list of tasks.
GET_TEMPLATE_LIST	Returns a list of all templates in the system
IMPORT_OBJECTS	Acts as an interface to the LANSAs Import Facility.
OBJECT_PROPAGATE	Propagate an object to a given repository group.
PACKAGE_CREATE	Creates a package based on the supplied Deployment Tool Template.
PACKAGE_BUILD	Builds a package that has been defined using the Deployment Tool or the PACKAGE_CREATE Built-In Function.

Field and Component Related Built-In Functions

Built-In Function	Description
DLT_FIELD	Deletes a field definition from the LANSAs Repository.
GET_COMPONENT_LIST	Returns a list of Components.
GET_FIELD	Retrieves attributes of a LANSAs Repository field.
GET_FIELD_INFO	Retrieves a list of LANSAs internal database field information
GET_FIELD_LIST	Retrieves a list of LANSAs Repository fields and descriptions.
GET_HELP	Gets a list of help text for a specified field, function or process.
GET_MULTVAR_LIST	Retrieves a list of multilingual variables and values.
GET_ML_VARIABLE	Retrieves a multilingual variable definition.
GET_SYSVAR_LIST	Retrieves a list of system variables and descriptions, programs and program types.
GET_SYSTEM_VARIABLE	Retrieves a system variable definition.
PUT_FIELD	Inserts/updates a field in the LANSAs Repository.
PUT_FIELD_ML	Puts/updates a list of field multilingual attributes in different languages.
PUT_HELP	Puts/updates a list of help text for a specified field, function or process.
PUT_ML_VARIABLE	Adds/updates a multilingual variable definition to the Repository.
PUT_SYSTEM_VARIABLE	Creates/amends a system variable.

File Related Built-In Functions

Built-In Function	Description
ACCESS_FILE	Read records from any file in the system.
ACCESS_RTE	Specifies/re-specifies the attributes of an "access route".
ACCESS_RTE_KEY	Specifies/re-specifies the name of a field or value that is to be used to access data via an access route.
COMPARE_FILE_DEF	Compares two CTD files and indicates if a difference is found.
DLT_FILE	Submits a batch job to delete a file and its associated logical files and I/O module.
END_FILE_EDIT	Ends an "edit session" on the definition of a nominated LANSAs file.
FILE_FIELD	Specifies/re-specifies a field in format of the file being edited.
FILE_FIELD VIRTUAL	Specifies/re-specifies a virtual field of the file definition being edited.
GET_FILE_INFO	Retrieves a list of file related information from the LANSAs internal database.
GET_LOGICAL_LIST	Retrieves a list of physical files, associated logical views and descriptions.
GET_PHYSICAL_LIST	Retrieves a list of physical files and descriptions.
LOAD_FILE_DATA	Calls the OAM for the requested file and loads the data.
LOAD_OTHER_FILE	Loads the definition of an "OTHER" file.
LOGICAL_KEY	Specifies/re-specifies the name of a field that is a key of a logical view/file.
LOGICAL_VIEW	Specifies/re-specifies the name and basic

attributes of a logical view/file

MAKE_FILE_OPERATIONNL	Submits a batch job to create/recreate a file plus associated logical files and I/O module.
PHYSICAL_KEY	Specifies/re-specifies the key of the physical file associated with the file being edited.
PUT_FILE_ML	Puts/updates a list of file multilingual attributes in different languages.
REBUILD_FILE	Optionally drops the existing file and its views, and creates the new file from the CTD file .
REBUILD_TABLE_INDEX	Rebuild IBM i High Speed Index
RESET_@@UPID	Reset the @@UPID field to zero in any file.
SET_FILE_ATTRIBUTE	Sets a file's database attributes.
START_FILE_EDIT	Starts an "edit session" on the definition of a nominated LANSAs file definition.
STM_FILE_CLOSE	Closes the stream file which was opened by a STM_FILE_OPEN
STM_FILE_OPEN	Opens a stream file.
STM_FILE_READ	Reads data from the specified stream file that was opened by STM_FILE_OPEN.
STM_FILE_WRITE	Writes data to the specified stream file that was opened by STM_FILE_OPEN.
STM_FILE_WRITE_CTL	Writes Line terminator character/s to the data stream.
UNLOAD_FILE_DATA	Calls the OAM for a file and unloads all its data to a flat file.

Function Related Built-In Functions

Built-In Function	Description
DELETE_FUNCTION	Deletes all details of the function currently being edited.
END_FUNCTION_EDIT	Ends an active edit session on a LANSAs function definition.
GET_FUNCTION_ATTR	Gets an attribute of the function being edited.
GET_FUNCTION_INFO	Retrieves a list of function related information from the LANSAs internal database for the RDML function.
GET_FUNCTION_LIST	Retrieves a list of processes, associated functions and descriptions from the LANSAs internal database.
GET_FUNCTION_RDML	Returns the RDML code associated with a function.
PUT_FUNCTION_ATTR	Sets an attribute of a function definition that is being edited.
PUT_FUNCTION_ML	Puts/updates a list of function multilingual attributes in different languages.
PUT_FUNCTION_RDML	Stores the RDML code associated with a function from a working list.
START_FUNCTION_EDIT	Starts an "edit session" on the definition of a nominated LANSAs function definition.

LANSA Composer Built-In Functions

Built-In Function	Description
COMPOSER_CALLF	Calls a named LANSAs function.
COMPOSER_USE	Runs a LANSAs Composer Processing Sequence.
COMPOSER_RUN	Connects to the nominated LANSAs or LANSAs Composer server system.

LANSA Integrator Built-In Functions

Built-In Function	Description
MQSeries in the <i>LANSA Integrator Guide</i>	Transfer messages from a message queue to loaded service for processing. Transfer XML responses from the loaded service into a message queue.

Java Service Manager Built-In Functions

Built-In Function	Description
JSM_COMMAND	Sends a command string to the currently open JSM server connection.
JSM_OPEN	Opens a connection to the JSM server.
JSM_CLOSE	Closes the currently open connection to the JSM server.
JSMX_COMMAND	Sends a command string to the JSM connection identified by the handle.
JSMX_OPEN	Opens a connection to the JSM server to start a service thread for commands sent by JSMX_COMMAND.
JSMX_CLOSE	Closes the JSM connection identified by the handle.

List Handling Built-In Functions

Built-In Function	Description
<code>DELETE_SAVED_LIST</code>	Delete a previously saved list.
<code>RESTORE_SAVED_LIST</code>	Restore a previously saved list.
<code>SAVE_LIST</code>	Save a working list to disk.
<code>TRANSFORM_FILE</code>	Transform a disk file into a working list(s).
<code>TRANSFORM_LIST</code>	Transform working list(s) to a disk file.

Locking Built-In Functions

Built-In Function	Description	LANSA for i
LOCK_OBJECT	Lock a User Object.	Y
UNLOCK_OBJECT	Unlock a User Object.	Y

Mathematical Built-In Functions

Built-In Function	Description
EXPONENTIAL	Calculate exponential value.
RANDOM_NUM_GENERATOR	Returns a random number between 0 and 1.
ROUND	Rounds off a decimal value.
SQUARE_ROOT	Calculate a square root value.

Messages and Message Handling Built-In Functions

Built-In Function	Description
CLR_MESSAGES	Clear all messages from RDML program queue.
GET_MESSAGE	Gets details of next message from RDML pgm queue.
GET_MESSAGE_DESC	Gets the description of a message from a msg file.
GET_MESSAGE_LIST	Loads the list with each subsequent message file/message stored in the message table.
ISSUEINQUIRY	Issue an inquiry message to a message queue.
ISSUEMESSAGE	Issue a message to a message queue.
MESSAGE_COLLECTOR	Nominate function as a message collector.

Message Box Built-in Functions

Built-In Function	Description
<code>MESSAGE_BOX_ADD</code>	Adds items to the message box assembly area.
<code>MESSAGE_BOX_APPEND</code>	Appends items to the message box assembly area.
<code>MESSAGE_BOX_CLEAR</code>	Clears the current message box assembly area.
<code>MESSAGE_BOX_SHOW</code>	Causes a standard MS message box to be displayed.

Miscellaneous Built-In Functions

Built-In Function	Description
GET_IENTRY_LIST	Retrieves a list of Impact List entries and descriptions from the data dictionary.
LIST_PRINTERS	Retrieves a list of printers currently configured on the machine.
MAKE_SOUND	Causes a standard sound to be queued.

Other Vendor Built-In Functions

Built-In Function	Description
------------------------------	--------------------

Other Vendor Built-In Functions	A collection of miscellaneous Built-In Functions created by LANSAs users and distributed by LANSAs as a service. These Built-In Functions carry no guarantees and you use them at your own risk.
---	---

Process Related Built-In Functions

Built-In Function	Description
COMPILE_PROCESS	Submits a batch job to compile a process and all selected functions.
COMPILE_COMPONENT	Compiles a component.
DELETE_PROCESS	Submits a batch job to delete a process and all of its functions.
DLT_PROCESS_ATTACH	Deletes attached processes and/or functions from the definition of the process definition being edited.
END_PROCESS_EDIT	Ends an active edit session on a LANSAs process definition.
GET_PROCESS_ATTR	Gets attributes of a process definition.
GET_PROCESS_INFO	Retrieves a list of process related information from the LANSAs internal database.
GET_PROCESS_LIST	Retrieves a list of processes and descriptions from the LANSAs internal database.
PUT_PROCESS_ACTIONS	Puts the definition of an action bar layout into the definition of the process definition.
PUT_PROCESS_ATTACH	Puts a process and/or function "attachment" into the definition of the process definition.
PUT_PROCESS_ATTR	Sets an attribute of a process definition within an edit session.
PUT_PROCESS_ML	Puts/updates a list of process multilingual attributes in different languages.
START_PROCESS_EDIT	Starts an "edit session" on the definition of a nominated LANSAs process definition.

Rule/Trigger Related Built-In Functions

Built-In Function	Description
DELETE_CHECKS	Deletes standard Repository or File level validation checks from nominated field.
DELETE_TRIGGERS	Deletes standard Repository or File level triggers from nominated field for subsequent replacement
PUT_COND_CHECK	Create/amend a "simple conditional logic" Repository or File level validation check
PUT_DATE_CHECK	Create/amend a "date range/date format" Repository or File level validation check.
PUT_FILE_CHECK	Create/amend a "code/table file lookup" Repository or File level validation check.
PUT_PROGRAM_CHECK	Create/amend a "call user program" Repository or File level validation check.
PUT_RANGE_CHECK	Create/amend a "range of values" Repository or File level validation check.
PUT_TRIGGER	Create/amend a Repository or File level trigger.
PUT_VALUE_CHECK	Create/amend a "list of values" Repository or File level validation check.

Space Built-In Functions

Built-In Function	Description
CREATE_SPACE	Creates a space object with the specified name
DEFINE_SPACE_CELL	Defines a cell (or column) within the specified space object.
DELETE_IN_SPACE	Deletes all cell rows that matches the key values supplied
DESTROY_SPACE	Destroys a space object with the specified name
FETCH_IN_SPACE	Retrieves the values of the first cell row that matches the key values supplied.
INSERT_IN_SPACE	Inserts a set of cell values into a space object
SELECT_IN_SPACE	Retrieves the values of the first row of cells that match the key values supplied
SELECTNEXT_IN_SPACE	Retrieves the values of the next row of cells that match the key values supplied.
SPACE_OPERATION	Request a miscellaneous space object operation
UPDATE_IN_SPACE	Updates a single cell row that matches the key values supplied

Spool File Built-In Functions

Built-In Function	Description
START_RTV_SPLF_LIST	Provides the selection criteria for retrieval of spool files.
GET_SPLF_LIST_ENTRY	Retrieves the details of the specified spooled files.
END_RTV_SPLF_LIST	Closes the list and releases the storage allocated to that list.

String Handling Built-In Functions

Built-In Function	Description
BCONCAT	Concatenate strings with a blank character.
BINTOHEX	Converts source field from binary to alphanumeric.
CENTRE	CENTRE an argument string into a return string.
CHECKNUMERIC	Check and convert an alpha string to numeric form.
CHECKSTRING	Check a string only contains allowable values.
CONCAT	Concatenate strings in full (no truncation).
CONVERT_STRING	Converts a text string from one encoding to another.
DECRYPT	Decrypts a string.
ENCRYPT	Encrypts a string.
FILLSTRING	Fill a field with occurrences of a string.
FORMAT_STRING	Returns a character string which is built from an input Format Pattern.
GET_KEYWORD_STRING	Get keywords and values from an ESF string.
HEXTOBIN	Converts source field from alphanumeric to binary.
LEFT	Left align argument into return string.
NUMERIC_STRING	Convert a number to a string.
REVERSE	Reverse a text string.
RIGHT	Right align argument into return string.
SCANSTRING	Scan a string for first occurrence of a pattern.
TCONCAT	Concatenate strings with trailing blanks truncated.

UPPERCASE

Convert a string to uppercase characters.

Template Related Built-In Functions

Built-In Function	Description
<code>EXECUTE_TEMPLATE</code>	Executes an application template to generate RDML function code into a working list.
<code>TEMPLATE_@@ADD_LST</code>	Allows a new field to be added to an application template list.
<code>TEMPLATE_@@CANSNNN</code>	Allows an application template character reply variable to be set before execution.
<code>TEMPLATE_@@CLR_LST</code>	Clears an application template list.
<code>TEMPLATE_@@GET_FILS</code>	Returns a list of all related files from a nominated base file
<code>TEMPLATE_@@NANSNNN</code>	Allows an application template numeric reply variable to be set before execution.
<code>TEMPLATE_@@SET_FILS</code>	Allows file(s) from a list of files to be "selected" for use within an application template.
<code>TEMPLATE_@@SET_IDX</code>	Allows an index variable to be set to a nominated value before execution.

Unique Operating System and Platform Access Built-In Functions

Built-In Function	Description
DLL	Invoke a function contained in a DLL.
GET_REGISTRY_VALUE	This BIF will return the Value for the specified Registry Key.
PUT_REGISTRY_VALUE	This BIF will add/update the value for the specified Registry Key.
SYSTEM_COMMAND	Execute an operating system command.

Web Built-In Functions

Built-In Function	Description
DELETE_WEB_COMPONENT	Deletes a Web Component and backups.
GET_WEB_COMPONENT	Gets the page text for a Web Component.
PUT_WEB_COMPONENT	(Re)build a Web Component. Use when generating a hand crafted Web Component at execution time.

Zip Built-In Functions

Also see

[Zip Built-in Function Note](#)

Built-In Function Description

ZIP_ADD	Adds files to zip file.
ZIP_DELETE	Deletes files from zip file.
ZIP_EXTRACT	Extracts files from zip file.
ZIP_GET_INFO	Retrieve information about zip file.
ZIP_MAKE_EXE	Create executable file from zip file.

System Variables

Refer to:

- [General Variables](#)
- [Function Only Variables](#)
- [BIF Variables](#)
- [Special Variables](#)
- [SuperServer Variables](#)
- [System Variable Evaluation Programs](#) in the *Visual LANSA Developer Guide*.

Formats, Values and Codes

Refer to:

- [Date Formats](#)
- [Standard Field Edit Codes](#)
- [Arithmetic and Expression Operators](#)
- [RDML Field Attributes and their Use](#)
- [RDML I/O Return Codes](#)
- [Help Text Enhancement and Substitution Values](#)

1. Fields

Fields can be defined in either:

- the LANSAs Repository or
- directly in the RDML Source Code using the [DEFINE](#) command.

The information in this chapter relates to fields defined in the LANSAs Repository.

Specific details of field type rules, warnings, tips and techniques, and platform considerations are documented by field type.

Also See

[1.1 Field Types](#)

[1.2 Field Definitions](#)

[1.3 Field Visualizations](#)

[1.4 Field Rules and Triggers](#)

[1.5 Field Help Text](#)

[Components](#)

[Fields](#) in the *User Guide*

[Creating Fields](#) in the *Developer Guide*

1.1 Field Types

Please review the general [1.1.1 Field Type Considerations](#). Specific details of field type rules, warnings, tips and techniques, and platform considerations are documented by field type. Select the field type of interest:

1.1.2 Alpha	1.1.11 Date	1.1.4 NVarChar
1.1.14 Binary	1.1.13 DateTime	1.1.7 Packed
1.1.17 BLOB	1.1.16 DBCS Graphic String	1.1.8 Signed
1.1.19 Boolean	1.1.10 Float	1.1.3 String
1.1.5 Char	1.1.9 Integer	1.1.12 Time
1.1.18 CLOB	1.1.6 NChar	1.1.15 VarBinary

Also See

[1.2 Field Definitions](#)

[↑ 1. Fields](#)

1.1.1 Field Type Considerations

Field Type Use Recommendations	Field Type Conversions	RDMLX Enabled Partition	SQL Null Handling	What Classifies a Field as RDML?
--	--	---	-----------------------------------	--

Field Definitions

- Field types [1.1.2 Alpha](#), [1.1.7 Packed](#), and [1.1.8 Signed](#) can be used in all partitions.
- Field types [1.1.11 Date](#), [1.1.12 Time](#), [1.1.13 DateTime](#), [1.1.14 Binary](#), [1.1.15 VarBinary](#), [1.1.3 String](#), [1.1.4 NVarChar](#), [1.1.6 NChar](#), [1.1.5 Char](#), [1.1.9 Integer](#), [1.1.10 Float](#), [1.1.17 BLOB](#), [1.1.18 CLOB](#) and [1.1.19 Boolean](#) must be used in an [RDMLX Enabled Partition](#). Refer to [RDML and RDMLX Partition Concepts](#) in the *Administrator Guide* for details about these partition differences.
- Packed, Signed and Integers allow Edit Codes and Edit Words. No other field types support Edit Codes or Edit Words.
- Keyboard Shift Blanks is the **only** valid value for Binary, VarBinary, Date, Time, DateTime, Integer, Float, NVarChar, NChar, BLOB and Boolean.
- In an RDMLX partition, a field that is defined by a reference field must not have different attributes to the reference field, regardless of the *IMPREFFLDNOPROP flag in the system definitions.
- ASQN attribute (see Common Attributes in [1.2.17 Input Attributes](#)) will be enabled by default for new fields of types Date, Time, DateTime, Binary, VarBinary, BLOB, and CLOB.

Field Usage in LANSAs

- LANSAs has implemented strong typing for all field types.
- Alpha, String and Char are all classed as **String** types and are valid for LANSAs arguments of type 'A'.
- Packed, Signed, Float and Integer are classed as **Numbers** and so are valid for LANSAs arguments of type 'N'.
- All other types like Date, DateTime and BLOB are classed as their own type and are not valid for either a type 'A', or type 'N'.
- Use of Hex 00 is not supported by LANSAs in Alpha, String, Char, NVarChar

and NChar fields. The behavior will vary depending on what platform you are on and the database you write the data to. The use of hex 00 in Alpha working fields where overlays are used may have a valid purpose in RDML on IBM i, but it is up to the you to ensure that the LANSAs behavior meets the needs of the application on all platforms and databases where the application may run.

- Field types have a default property, usually **.Value**, and additional properties as required. For a Date field #MyDate, the following examples all have an identical meaning:

```
Define Field(#MyDate) Type(*DATE)
Change #MyDate To(1972-03-04)
Set #MyDate Value(1972-03-04)
#MyDate.Value := "1972-03-04"
```

- Rather than the default property **.Value**, fields of type BLOB and CLOB have a default property called **.FileName** to clearly indicate that changing the "value" of the field is actually changing its default property which is a file name property.
- Fields that are SQL-nullable can be tested with *IS or *ISNOT *SQLNULL, and also have an intrinsic property called .IsSqlNull. This can be used to determine the current state of the field. For example:

```
If (#Std_Num.IsSqlNull)
```

```
If (#Std_Num *ISNOT *SqlNull)
```

```
If ((*Not #Std_Num.IsSqlNull) *AndIf (#Std_Num > 45.00))
```

```
If ( ([your expression] *IS *SqlNull )
```

Built-In Function Argument/Return Value Types

- Refer to [BIF Function Rules](#) for information.

Tips & Techniques

- LANSAs has implemented strong typing for all field types. For example, a Built-In Function that allows an Alpha will also support String. It will not allow a DateTime. To use a DateTime, you simply use #MyDateTim.asString, explicitly stating that you want to use the string representation.
- If you are using RDML Partitions, you may wish to refer to the *LANSAs for i User Guide*. This documentation has been written for development using RDML Partitions only.

- Some field types such as [1.1.10 Float](#) are primarily supported in LANSAs for use with Other Files and are not recommended for use in business applications. It is recommended that you use [1.1.9 Integer](#) or [1.1.7 Packed](#) fields for the majority of your numeric data, and [1.1.8 Signed](#) is also recommended for RDML partitions.
- Character data can be stored in [1.1.2 Alpha](#), [1.1.3 String](#), [1.1.5 Char](#), [1.1.4 NVarChar](#) and [1.1.6 NChar](#) type fields. Where possible, use of [1.1.2 Alpha](#) fields is recommended.
- Select the vendor specific DBMS you are targeting and be sure to review all warnings when creating/editing files to ensure that there are no problems supporting a specific field range for the selected DBMS.

Platform Considerations

- When creating a LANSAs file, LANSAs field types are converted to specific DBMS Data types supported by each vendor. For example, an Integer field (depending upon its length) might be implemented as a TINYINT, SMALLINT, INTEGER, BIGINT, NUMBER(3), NUMBER(5), or other type, depending upon the specific vendor DBMS layer supported. The file x_dbmenv.dat controls this behavior though it must not be modified.
- In some cases, a particular vendor's DBMS may not be able to handle the full length of a field because the length of data available in LANSAs exceeds the length available in the vendor's database. It is possible to select the target DBMS system and warnings will be displayed where the field may not be preserved exactly. In these cases, the data will either be truncated, or rejected. For example, if you define a DateTime of length 29, Oracle supports storing the full 9 decimal places. Other DBMSs might automatically truncate data to 0, 3, or 6 decimal places.

Also See

[1.1.2 Alpha](#) [1.1.9 Integer](#) [1.1.15 VarBinary](#) [1.1.8 Signed](#)
[1.1.11 Date](#) [1.1.17 BLOB](#) [1.1.3 String](#) [1.1.13 DateTime](#)
[1.1.14 Binary](#) [1.1.7 Packed](#) [1.1.10 Float](#) [1.1.17 BLOB](#)
[1.1.5 Char](#) [1.1.12 Time](#) [1.1.18 CLOB](#) [1.1.19 Boolean](#)
[1.1.6 NChar](#) [1.1.4 NVarChar](#)

[↑ 1.1 Field Types](#)

Field Type Use Recommendations

If you want to define a new field into the LANSAs dictionary use this decision tree:

- Is the field a Date or a Time?
 - To store a unique day use type **Date**.
 - To store a unique moment in time use type **DateTime** (in preference to two separate Date and Time fields).
 - To store a non-unique time, which is unusual, use **Time**.
- Is it a True/False value? If so, use **Boolean**.
- Is it a number? If there are no decimal places use type **Integer** otherwise use type **Packed**.
- Then it must be a string of characters or some other stream of bytes.
 - Is it normally kept in a specialized type of file or document (eg: an image, a sound, an MS-Word document or an XML document)? Use type **BLOB**.
 - Do you expect it to be subjected to normal code page conversions? If not, use **VarBinary** unless your data is always the same length, in which case you may choose to use **Binary**.
 - Will the field be 256 bytes or less and be best stored as a fixed length field? If yes, and it only needs to support one language at execution time, use type **Alpha**.
 - Will it ever be longer than 65535 bytes? Use type **CLOB**.
 - Do you store the field as variable length with any trailing blanks significant (for example, included in field concatenation operations)? If it only needs to support one language at execution time, use type **String**. Otherwise, use type **NVarChar**.
 - Otherwise it is fixed length. If it only needs to support one language at execution time, use type **Char**. Otherwise, use type **NChar**.

All other types are provided for compatibility with non-LANSAs database tables. Generally you should avoid using them when defining new fields or columns in new DBMS tables.

Commonly Used Field Types

Type	Description	Typical DBMS Storage Type	Maximum Dictionary Definable Length	Subject to Code Page Conversions	Notes
Integer	Integer	INTEGER	4 bytes.		Hardware and compile behavior may vary
Packed	Standard decimal number	DECIMAL	RDMLX programs: 63 digits of which up to 63 may be decimals RDML programs: 30 digits of which up to 9 may be decimals	N/A	
String	Variable length alphanumeric string with a dictionary defined maximum length.	VARCHAR (variable length)	65535	YES	Not padded with blanks to the dictionary defined maximum length, except that a zero

					length string has 1 space added.
VarBinary	Exactly the same as String but never subjected to code page conversions.	VARBINARY	32767	NO	Not padded to the dictionary defined maximum length.
NVarChar	Similar to String but data is handled as Unicode. This allows multiple languages to be used at execution time.	NVARCHAR	65535 characters	YES (but only when converted to native String)	
Alpha	Constant length alphanumeric string with a dictionary defined maximum length.	CHAR (fixed length)	256	YES	Always padded with blanks to the dictionary defined maximum length.
Date	Date in ISO format: YYYY-MM-DD	DATE	N/A	N/A	
Time	Time in ISO format HH:MM:SS	TIME	N/A	N/A	
DateTime	Date and Time in ISO format YYYY-MM-DD	DATETIME	29	N/A	

	HH:MM:SS[.ffffff]				
	where the existence and length of the [.ffffff] portion are definable.				
Boolean	True/False	Decimal	N/A	N/A	

Specialized Character and Binary Types (useable only in RDMLX programs)

Type	Description	Typically Stored in a DBMS as	Maximum Dictionary Definable Length	Typical DBMS Storage length used	Subject to Code Page Conversions	Important
Char	Exactly the same as String, except fixed length and DBMS storage implementation is usually CHAR.	CHAR	65535	Fixed length	YES	
Binary	Exactly the same as VarBinary but DBMS implementation varies.	BINARY	32767	May be either Fixed or Variable.	NO	Databases that use fixed length column types will pad to the maximum length.
NChar	Exactly the same as NVarChar, except fixed	NCHAR	65535	Fixed length	YES (but only when converted to native	

	length and DBMS storage implementation is usually NCHAR				String)	
CLOB	Character Large Object	CLOB	Undefined	Varies according to content	YES	DBMS performance consideration may apply.
BLOB	Binary Large Object	BLOB	Undefined	Varies according to content	NO	DBMS performance consideration may apply.

Rarely Used Numeric Types

Type	Description	Typical DBMS storage Type	Maximum Dictionary Definable Length	When to Use	Important
Signed	Signed or Zoned decimal	DECIMAL	RDMLX programs: 63 digits of which up to 63 may be decimals RDML programs: 30 digits of which up to 9 may be decimals	For programmatic numeric/character overlaying operations.	Signed is less efficient than Packed or Integer on mathematical operations.
Float	Floating Point	FLOAT	Undefined. Floating point	When using non-LANSA defined	Hardware and compiler

			numbers are approximations, not exact numbers.	DBMS tables only.	behaviors may vary.
--	--	--	--	-------------------	---------------------

[↑ 1.1.1 Field Type Considerations](#)

LOB 4	Yes	Yes ⁵	No	No	Yes	Yes	No
Boolean	Yes	Yes	Yes	No	No	No	Yes

Table notes:

1. CLOBs are treated as file pointers, so do not support conversion to a Numeric or Date/Time field type.
2. No support for conversion
 - between Integer or Float and the Date/Time field types
 - of a numeric type directly to a DateTime, as there is no way to determine if the type should be setting the Date or Time portion of the DateTime.
3. DateTime -> Date and DateTime -> Time are obvious.
 - The default Date is 1900-01-01 and the default time is midnight (00:00:00). Therefore:
 - Date -> DateTime sets the time part to midnight
 - Time -> DateTime sets the date part to 1900-01-01
 - Date -> Time always results in midnight
 - Time -> Date always results in 1900-01-01
4. The "value" for a LOB is a filename. Thus, when converting it is the filenames that are being copied, not the data. If an invalid filename is copied to a LOB, an error will occur at some stage.
5. When converting a Unicode string to a native string, the intrinsic .AsString may need to be used.

[↑ 1.1.1 Field Type Considerations](#)

RDMLX Enabled Partition

Please refer to the **Field Type** information in the [RDMLX Partition Settings](#) in the *Administrator Guide* for details about enabling manual field creation in the partition. Also refer to the Field Definition option [1.2.14 Enable Field for RDMLX](#).

Note: If a field type is not enabled in a partition, the field type may still be used if loaded as an OTHER file. Developers are not allowed to manually create fields of the type that has been disallowed. Hence, you may see a field type (loaded from an other file) in the repository, but you may not be allowed to manually create that field type.

Also See

[RDML and RDMLX Partition Concepts](#) in the *Administrator Guide*.

[↑ 1.1.1 Field Type Considerations](#)

SQL Null Handling

SQL Null support has been introduced mainly for Other File integration.

SQL Null indicates that data is unknown. For example, the field #BirthDate could be SQL Null for a record read from file Person, indicating that we do not have the person's birthdate. Without SQL Null support, the only alternative is to set the date something unlikely (such as 0001-01-01).

A field must have the [ASQN \(Allow SQL Nulls\) attribute](#) to utilize SQL Null functionality such as the [*SQLNULL Keyword](#), the [Intrinsic Property .IsSqlNull](#), and the [Intrinsic Method .AsValue](#).

At execution time, a field that allows SQL Null either has a real value or is SQL Null. Fields allowing SQL Null may behave differently at execution time when they are SQL Null, and this behaviour is also dependent on whether the function option [*STRICT_NULL_ASSIGN](#) has been specified. Refer to [Assignment, Conditions, and Expressions with Fields allowing SQL Null](#) for further details.

When working with Other Files, behaviour for SQL Null fields may vary on Insert and Update, if the database definition of the column differs from the LANSAs definition. Refer to [UPDATE Comments/Warnings](#) for further details.

[↑ 1.1.1 Field Type Considerations](#)

ASQN (Allow SQL Nulls) attribute

ASQN is a field storage attribute along the lines of SBIN and SREV. It may be specified as either an input or output attribute. A field must have the ASQN attribute to utilize SQL Null functionality.

The ASQN attribute is only available when the partition has been enabled for full RDMLX. Fields using the ASQN attribute are RDMLX fields.

Also See

[*SQLNULL Keyword](#)

[Intrinsic Property .IsSqlNull](#)

[Intrinsic Method .AsValue](#)

[↑ SQL Null Handling](#)

***SQLNULL Keyword**

*SQLNULL is a special keyword like *NULL, but with a different meaning: *SQLNULL means the value for the field is unknown, whereas *NULL means the field is empty.

Fields may not be compared to *SQLNULL with operators such as *EQ and *NE, however *IS and *ISNOT can be used.

*SQLNULL becomes the default for fields that allow SQL Null, regardless of field type, unless another default has been specified. *SQLNULL may be explicitly specified as the default for a field if it has the ASQN attribute. For example, in RDML:

```
Define Field(#B) Type(*CHAR) Length(8) Input_Atr(ASQN)
Default(*SQLNULL)
```

Also See

[ASQN \(Allow SQL Nulls\) attribute](#)

[Intrinsic Property .IsSqlNull](#)

[Assignment, Conditions, and Expressions with Fields allowing SQL Null](#)

[↑ SQL Null Handling](#)

Intrinsic Property .IsSqlNull

The ASQN attribute has an intrinsic read-only property **.IsSqlNull**. This may have the values True or False. If the field is SQL Null, .IsSqlNull is True, otherwise it is False.

Note that Intrinsic properties can also be used in conditions that are defined by or passed to an I/O module, but this is not recommended. Refer to [Specifying WHERE Parameters in I/O Commands](#).

Also See

[ASQN \(Allow SQL Nulls\) attribute](#)

[Intrinsic Method .AsValue](#)

[Assignment, Conditions, and Expressions with Fields allowing SQL Null](#)

[↑ SQL Null Handling](#)

Intrinsic Method .AsValue

The ASQN attribute has an intrinsic method **.AsValue**. If you want to control what value will be returned from an expression if the value of a field is SQL Null, use the AsValue() intrinsic method for the field. The AsValue parameter will be returned instead of SQL Null when the field's value is SQL Null.

Also See

[ASQN \(Allow SQL Nulls\) attribute](#)

[Intrinsic Property .IsSqlNull](#)

[Assignment, Conditions, and Expressions with Fields allowing SQL Null](#)

[↑ SQL Null Handling](#)

Assignment, Conditions, and Expressions with Fields allowing SQL Null

This section described the behavior of SQL Null fields in

- [Assignment](#),
- [Conditions](#) and
- [Expressions](#).

The following field definitions are used throughout the text:

```
Define Field(#A) Type(*DEC) Length(9) Decimals(0) Input_Atr(ASQN)
Default(*SQLNULL)
```

```
Define Field(#B) Type(*DEC) Length(9) Decimals(0) Input_Atr(ASQN)
Default(*SQLNULL)
```

```
Define Field(#C) Type(*DEC) Length(9) Decimals(0) Default(*NULL)
```

Also See

[ASQN \(Allow SQL Nulls\) attribute](#)

[Specifying Conditions and Expressions](#) in the *Technical Reference Guide*.

Assignment

If you wish to ensure that SQL Null fields are handled via ANSI rules for assignment, enable the `*STRICT_NULL_ASSIGN` function option. This option causes a fatal error to occur at execution time if the source field is SQL Null and the target field does not have the ASQN attribute.

A field allowing SQL Null may be explicitly set to SQL Null, as in the following example.

```
#B := *SQLNULL
```

A field that is currently SQL Null may be assigned to another field. If the target field allows SQL Null, it will be set to SQL Null. In the following example, #A becomes SQL Null because #B was SQL Null.

```
#A := #B
```

If the target field does not have the ASQN attribute, the behavior varies depending on whether the `*STRICT_NULL_ASSIGN` function option is

enabled. By default, the *NULL value for the field type will be assigned to the target field. In the example below, as #C is a numeric field, it would be set to zero. For a definition of what the *NULL value is for each of the field types refer to [CHANGE Parameters](#).

```
#C := #B
```

However, if *STRICT_NULL_ASSIGN has been enabled, and the example code above is executed when #B is SQL Null, a fatal error will occur as the target field does not support being set to SQL Null. When working with *STRICT_NULL_ASSIGN, the LANSAs Developer must code carefully to protect against such runtime errors. For example:

```
If (*Not #B.IsSqlNull)
#C := #B
Else
Message Msgtxt('#B is SQL Null')
Endif
```

You can also use the .AsValue intrinsic method to treat an SQL Null field as a different value. This is useful for mathematics and concatenation, where SQL Null or *NULL are not appropriate values. In the following example, we now get the result of 5 in #C if #B is SQL Null. However, if #B was 3, #C would be set to 15 because #B.AsValue only affects #B when it is SQL Null.

```
#C := #B.AsValue( 1 ) * 5
```

Also See

[*SQLNULL Keyword](#)

[Intrinsic Property .IsSqlNull](#)

[Intrinsic Method .AsValue](#)

Conditions

- To test for SQL Null, you should use *IS *SQLNULL or *ISNOT *SQLNULL, or the [Intrinsic Property .IsSqlNull](#).
- When using IF_NULL or .IsNull, an SQL Null field will return FALSE.
- Since SQL Null does not represent a value, when using an equality operator such as *EQ, *LE, *GT to compare fields, and one of the factors of the compare is SQL Null, the comparison will produce an SQL Null. When

combined with *OR and *AND operators, an SQL Null factor will continue to produce an SQL Null. A conditional expression that produces an SQL Null will evaluate to false.

- SQLNULL comparisons will always stay as SQLNULL if the SQLNULL value is true. That is, when an expression is testing an SQLNULL, and there IS an SQLNULL, the expression will keep the SQLNULL value. For these types of scenarios, the *ORIF boolean feature should be used.

```
IF COND((#DATE2.IsSqlNull) *orif (#DATE1 *gt #DATE2))
#DATE2 := #DATE1
ENDIF
```

- If you want a condition to return TRUE for both Null and SQL Null, use the *ORIF boolean feature together with [Intrinsic Property .IsSqlNull](#) and [Intrinsic Property .IsNull](#). The following condition on our sample field #A will return true if the field is zero or SQL Null.

```
(#A.IsSqlNull) *orif (#A.IsNull)
```

- If you want a condition to return TRUE for both SQL Null and some other value, use the [Intrinsic Method .AsValue](#). The following condition will return true if the field is 1 or SQL Null.

```
#A.AsValue(1) *EQ 1
```

The following table summarizes the result of various conditions, with the sample fields #A and #B both SQL Null, and #C *ZERO.

Condition	Result
#A.IsSqlNull	TRUE
#A.IsNull	FALSE
(#A.IsSqlNull) *orif (#A.IsNull)	TRUE
IF_NULL(#A #B #C)	FALSE
IF_NULL(#C)	TRUE
#A.AsValue(*ZERO) *EQ *ZERO	TRUE
#A *EQ *ZERO	FALSE

#A *EQ #B	FALSE
#A *LE #B	FALSE
#B *EQ #C	FALSE
#B *LE #C	FALSE

Expressions

When expressions are being evaluated, intermediate results retain the SQL Null state. They are *ALWAYS* strictly interpreted. For example, when #B is SQL Null, the result of expression '#B + 1' is SQL Null. That is, SQL Null plus 1 is still SQL Null. This is independent of the attributes of any result field.

It is only when the result of the expression is assigned into the result field that a difference in behaviour can occur. If the result of an expression is SQL Null, behaviour depends on whether the result field allows SQL Null and also on the function option *STRICT_NULL_ASSIGN. Refer to [Assignment](#) for details.

If you wish to change the value of SQL Null fields to something more appropriate, use [Intrinsic Method .AsValue](#). For example, the result of expression '#B.AsValue(1) + 1' is 2 when #B is SQL Null (and 5 when #B is 4).

[↑ SQL Null Handling](#)

What Classifies a Field as RDML?

All working fields defined in RDMLX application code are classified as either RDML or RDMLX. If a field meets the following rules, it is an RDML field. Otherwise it is an RDMLX field.

- Field does not have the ASQN attribute.
- TYPE(*CHAR) LENGTH(n), where $n \leq 256$.
- TYPE(*DEC) LENGTH(n) DECIMALS(d), where $1 \leq n \leq 30$ and $0 \leq d \leq 9$
- TYPE(x) LENGTH(*REFFLD) REFFLD(#A), where x is *CHAR, *DEC, or *REFFLD and field #A is an RDML data dictionary field.

[↑ 1.1.1 Field Type Considerations](#)

1.1.2 Alpha

Alpha fields are used to store character type data up to a maximum of 256 characters in length.

Data may be SBCS, mixed SBCS and DBCS, or DBCS-only. Only the current native codepage is supported.

Alpha fields are classified as strings.

An Alpha does not treat trailing blanks as significant. They are truncated before concatenation and comparison.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining an Alpha field in the repository are:

Length Must be 1 to 256 in length. No decimals.

Valid Keyboard Shift Blanks, X, A, N, I, D, M, O, E, J, or W.

Allowed Attributes Null values are not allowed.

Edit Mask Not allowed.

Default Spaces.

Usage Rules

Partition Type RDML and RDMLX Partitions.

Files RDML and RDMLX Files. Alpha fields may be used as real fields or keys.

Logical Views Alpha fields may be used as keys to logical views.

Virtual Fields Alpha fields may be used with Substring, Concatenation or Date Virtual Fields.

Predetermined Join Fields Alphas may be used for lookup predetermined joined fields.

Built-In Functions When used in Built-In Functions, Alpha fields are valid for arguments of type 'A'.

Conversion Refer to [Field Type Conversions](#).

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Also See

[1.1.5 Char](#)

[1.1.3 String](#)

[↑ 1.1 Field Types](#)

1.1.3 String

String is a variable-length character field, with a maximum length between 1 and 65,535. (This is the byte length, not the number of characters.)

Strings store alpha data that is either SBCS, mixed SBCS and DBCS, or DBCS-only. Only the current native codepage is supported.

Strings are classified as strings.

A String retains any trailing blanks, they are significant. When concatenating a String with spaces on the end, those spaces are retained. But the space is NOT SIGNIFICANT for comparisons.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a String field in the repository are:

Length Strings may be 1 to 65,535 in length. Strings have no decimals.

Valid Blanks, O, E, J or W.

Keyboard
Shift

Allowed AB, ASQN, CS, DDNN, FE, FUNC, FUNU, FUNX, JNMC, JNMU,
Attributes JNMX, JNRC, JNRU, JNRX, LC, ND, PROC, PROU, PROX, RA,
RB, RL, RLTB, SREV, SUNI, USRC, USRU, USRX.

Edit Not allowed.

Mask

Default *NULL

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files Strings may only be used in RDMLX Files. Strings may be used as real or key fields. If used as keys, length must be no more than 4000 bytes.

Logical Views Strings may be used as key fields. If used as keys, length must be less than 4000 bytes.

Virtual Fields Strings may be used with Code Fragment Virtual Fields.

Predetermined Strings may be used for lookup predetermined joined fields.

Join Fields

RDML Commands DEF_HEAD, DEF_FOOT, DEF_BREAK, and DEF_LINE only support printing of RDML fields. Therefore, String fields need to be converted to Alpha to be used in reports.

Built-In Functions When used in Built-In Functions, Strings are valid for arguments of type 'A'.

Special Values *NULL, *NAVAIL, *REMEMBERED_VALUE

Conversion Refer to [Field Type Conversions](#).

Usage Notes

- Working fields may be defined as TYPE(*STRING).
- Fields of type **String** or **Char** and length of 256 or less may be used almost anywhere that fields of type Alpha may be used.
- A string of zero length has a space added to it before inserting or updating the database via SQL. This is in order to obtain consistent behaviour between our databases. Without a space, Oracle interprets the data as being SQL Null, which is not strictly true and is not how the other databases behave.

So, be aware that your application cannot make a distinction between an empty String and a String with 1 blank. Also, concatenations of a zero length String before inserting to the database and after reading may differ by the extra space. Behaviour can be made consistent by truncating trailing spaces before using Strings in an expression. Note that when comparing an empty String in RDML to a String with 1 space read from the database they will compare equal because trailing spaces are not significant in comparisons - they are only significant in expressions, like concatenation.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Also See

[1.1.2 Alpha](#)

[1.1.5 Char](#)

[1.1.4 NVarChar](#)

↑ 1.1 Field Types

1.1.4 NVarChar

NVarChar is a variable-length character field, with a maximum length between 1 and 65,535. (This is the number of characters, not the byte length.)

NVarChars store alpha data of any codepage. For example, in a list, an NVarChar field may have Japanese in one row, and French in another row.

NVarChars are classified as unicode strings.

An NVarChar retains any trailing blanks, they are significant. When concatenating an NVarChar with spaces on the end, those spaces are retained. But the space is NOT SIGNIFICANT for comparisons.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining an NVarChar field in the repository are:

Length	NVarChars may be 1 to 65,535 in length. NVarChars have no decimals.
Valid Keyboard Shift	Blanks.
Allowed Attributes	AB, ASQN, CS, FE, LC, ND, RA, RB, RL, RLTB, SREV.
Edit Mask	NVarChars may be used for lookup predetermined joined fields.
Default	*NULL

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files NVarChars may only be used in RDMLX Files. NVarChars may be used as real or key fields. If used as keys, length must be no more than 2000 characters.

Logical Views NVarChars may be used as key fields. If used as keys, length must be less than 2000 characters.

Virtual Fields NVarChars may be used with Code Fragment Virtual Fields.

Predetermined NVarChars may be used for lookup predetermined joined fields.

Join Fields

RDML Commands DEF_FOOT, DEF_BREAK, and DEF_LINE only support printing of RDML fields. Therefore, NVarChar fields need to be converted to Alpha to be used in reports.

Built-In Functions When used in Built-In Functions, NVarChars are valid for arguments of type 'A'.

Special Values *NULL, *NAVAIL,

Conversion Refer to [Field Type Conversions](#).

Usage Notes

- Working fields may be defined as TYPE(*NVARCHAR).
- A NVarChar of zero length has a space added to it before inserting or updating the database via Sql. This is in order to obtain consistent behaviour between our databases. Without a space, Oracle interprets the data as being SQL Null, which is not strictly true and is not how the other databases behave.

So, be aware that your application cannot make a distinction between an empty NVarChar and a NVarChar with 1 blank. Also, concatenations of a zero length NVarChar before inserting to the database and after reading may differ by the extra space. Behaviour can be made consistent by truncating trailing spaces before using NVarChars in an expression. Note that when comparing an empty NVarChar in RDML to an NVarChar with 1 space read from the database they will compare equal because trailing spaces are not significant in comparisons - they are only significant in expressions, like concatenation.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- If there is no need to support multiple languages at execution time, use an Alpha field or a String field.

Also See

[1.1.2 Alpha](#)

[1.1.3 String](#)

↑ 1.1 Field Types

1.1.5 Char

A Char is a fixed-length character field, with a length between 1 and 65,535. (This is the byte length, not the number of characters.) Char fields with lengths of 256 or less are equivalent to the existing Alpha field.

Char fields store alpha data that is either SBCS, mixed SBCS and DBCS, or DBCS-only. Chars are classified as strings.

Depending on the database type, Char may or may not treat trailing blanks as significant. If trailing blanks are not desired, a String field should be used.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Char field in the repository are:

Length Chars may be 1 to 65,535 in length. Chars have no decimals.

Valid Blanks, O, E, J or W.

Keyboard
Shift

Allowed AB, ASQN, CS, DDNN, FE, FUNC, FUNU, FUNX, JNMC, JNMU,
Attributes JNMX, JNRC, JNRU, JNRX, LC, ND, PROC, PROU, PROX, RA,
RB, RL, RLTB, SREV, SUNI, USRC, USRU, USRX.

Edit Not allowed.

Mask

Default *NULL

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files Chars may only be used in RDMLX Files. Chars may be used as real or key fields. If used as keys, length must be no more than 4000 bytes.

Logical Views Chars may be used as key fields. If used as keys, length must be less than 4000 bytes.

Virtual Fields Chars may be used with Code Fragment Virtual Fields.

Predetermined Chars may be used for lookup predetermined joined fields.

Join Fields

RDML Commands DEF_HEAD, DEF_FOOT, DEF_BREAK, and DEF_LINE only support printing of RDML fields. Therefore, Char fields need to be converted to Alpha to be used in reports.

Built-In Functions When used in Built-In Functions, Chars are valid for arguments of type 'A'.

Special Values *NULL, *NAVAIL, *REMEMBERED_VALUE

Conversion Refer to [Field Type Conversions](#).

Usage Notes

- Working fields defined with TYPE(*CHAR) where the field length is 256 or less should operate similarly to alpha fields.
- Fields of type **String** or **Char** and length of 256 or less may be used almost anywhere that fields of type Alpha may be used.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- If a field is a fixed character length of 256 or less, use an Alpha field.
- If you want trailing blanks to be significant, use a String field.

Also See

[1.1.2 Alpha](#)

[1.1.3 String](#)

[1.1.6 NChar](#)

[↑ 1.1 Field Types](#)

1.1.6 NChar

An NChar is a fixed-length character field, with a length between 1 and 65,535. (This is the number of characters, not the byte length.)

NChar fields store alpha data of any codepage. For example, in a list, an NChar field may have Japanese in one row, and French in another row.

NChars are classified as unicode strings.

Depending on the database type, NChar may or may not treat trailing blanks as significant. If trailing blanks are not desired, an NVarChar field should be used.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Char field in the repository are:

Length	NChars may be 1 to 65,535 in length. NChars have no decimals.
Valid Keyboard Shift	Blanks.
Allowed Attributes	AB, ASQN, CS, FE, LC, ND, RA, RB, RL, RLTB.
Edit Mask	Not allowed.
Default	*NULL

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files NChars may only be used in RDMLX Files. NChars may be used as real or key fields. If used as keys, length must be no more than 2000 characters.

Logical Views NChars may be used as key fields. If used as keys, length must be less than 2000 characters.

Virtual Fields NChars may be used with Code Fragment Virtual Fields.

Predetermined NChars may be used for lookup predetermined joined fields.
Join Fields

RDML DEF_HEAD, DEF_FOOT, DEF_BREAK, and DEF_LINE only

Commands	support printing of RDML fields. Therefore, NChar fields need to be converted to Alpha to be used in reports.
Built-In Functions	When used in Built-In Functions, Chars are valid for arguments of type 'A'.
Special Values	*NULL, *NAVAIL
Conversion	Refer to Field Type Conversions .

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- If there is no need to support multiple languages at execution time, use an Alpha field or a Char field.
- If you want trailing blanks to be significant, use an NVarChar field.

Also See

[1.1.2 Alpha](#)

[1.1.5 Char](#)

[1.1.4 NVarChar](#)

[↑ 1.1 Field Types](#)

1.1.7 Packed

Packed fields are exact, fixed-point numeric fields with a precision (# of significant digits), and a scale (# of digits after decimal place). The scale may be 0, indicating a whole number. Packed fields store signed numbers (+/-).

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Packed field in the repository are:

Length Packed fields have a maximum length of 63 in an RDMLX partition and 30 in an RDML partition. Maximum number of decimals is 63 in an RDMLX partition and 9 in an RDML partition.

Valid Keyboard Shift Blanks.

Allowed Attributes Null values are not allowed.

Edit Mask Use of an Edit Mask (that is, an Editcode or Editword) is allowed.

Default Zero.

Field Definition Notes

- Packed and Signed appear to be identical types, however a Packed field will almost always require less storage in memory (and in some databases) than type Signed.

Usage Rules

Partition Type RDML and RDMLX Partitions.

Files RDML and RDMLX Files. Packed fields may be used as real fields or keys.

Logical Views Packed fields may be used as keys to logical views.

Virtual Fields Packed fields may be used with Mathematical, Substring, Concatenation or Date Virtual Fields.

Predetermined Packed fields may be used for numeric predetermined joined

Join Fields fields.

Built-In Functions When used in Built-In Functions, packed fields are valid for arguments of type 'N'.

Conversion Refer to [Field Type Conversions](#).

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).
- Oracle and SQL Server support a maximum of 38 digits of which up to 38 may be decimals. You may use fields with a larger definition, but if these fields are on a file, the column will be created with a smaller size. For example, Packed(63,9) will be created as (38,9). If the field has more than 38 significant digits and you try to write it to the database, an error will occur.

Also See

[1.1.8 Signed](#)

[1.1.9 Integer](#)

[1.1.10 Float](#)

[↑ 1.1 Field Types](#)

1.1.8 Signed

Signed fields are exact, fixed-point numeric fields with a precision (# of significant digits), and a scale (# of digits after decimal place). The scale may be 0, indicating a whole number. Signed fields store signed numbers (+/-).

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Signed field in the repository are:

Length Signed fields have a maximum length of 63 in an RDMLX partition and 30 in an RDML partition. Maximum number of decimals is 63 in an RDMLX partition and 9 in an RDML partition.

Valid Blanks
Keyboard
Shift

Allowed Null values are not allowed.
Attributes

Edit Use of an Edit Mask (that is, an Editcode or Editword) is allowed.
Mask

Default Zero.

Field Definition Notes

- Packed and Signed appear to be identical types, however a Packed field will almost always require less storage in memory (and in some databases) than type Signed.

Usage Rules

Partition Type RDML and RDMLX Partitions.

Files RDML and RDMLX Files. Signed fields may be used as real fields or keys.

Logical Views Signed fields may be used as keys to logical views.

Virtual Fields Signed fields may be used with Mathematical, Substring, Concatenation or Date Virtual Fields.

Predetermined Signed fields may be used for numeric predetermined joined

Join Fields fields.

Built-In Functions When used in Built-In Functions, Signed fields are valid for arguments of type 'N'.

Conversion Refer to [Field Type Conversions](#).

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).
- Oracle and SQL Server support a maximum of 38 digits of which up to 38 may be decimals. You may use fields with a larger definition, but if these fields are on a file, the column will be created with a smaller size. For example, Packed(63,9) will be created as (38,9). If the field has more than 38 significant digits and you try to write it to the database, an error will occur.

Also See

[1.1.7 Packed](#)

[1.1.9 Integer](#)

[1.1.10 Float](#)

[↑ 1.1 Field Types](#)

1.1.9 Integer

Integers are whole numeric fields and are signed by default. Fields of type Integer have a length measured in bytes (1, 2, 4, 8). For example, a signed Integer of Byte Length 2, can store values from -32768 to +32767. Integers have no decimal places and are accurate. Integers are classed as Numbers in LANSAs.

The following table provides the implied length for each of the possible byte lengths for an Integer. The implied length is equivalent to the actual length of a signed or packed field.

# Bytes	Max value (signed)	Max value (unsigned)	Max # digits (implied length)
1	127	255	3
2	32767	65535	5
4	2147483647	4294967295	10
8	9223372036854775807	18446744073709551615	19 signed, 20 unsigned

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining an Integer in the repository are:

Length Integers may be 1, 2, 4, or 8 bytes in length. Integers have no decimals.

Valid Blanks

Keyboard

Shift

Allowed AB, ASQN, CS, FE, ND, RA, RB, RL, RLTB, SUNS

Attributes

Edit Use of an Edit Mask (i.e. an Editcode or Editword) is allowed if the

Mask Integer is signed. Integer does not allow edit codes W and Y. May use *DEFAULT.

Default *NULL

Field Definition Notes

- By default, Integers are signed.
- SUNS is only supported for one byte integers initially. If SUNS is enabled, it means the Integer cannot hold negative numbers, and so it will have a higher maximum value. An Integer(1) without SUNS ranges from -128 to +127, while an Integer(1) with SUNS ranges from 0 to 255.

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files Integers may only be used in RDMLX Files. Integers may be used as real fields or keys.

Logical Views Integers may be used as keys to logical views.

Virtual Fields Integers may be used with Mathematical Virtual Fields. Integers may be used as the source field(s).

Integers may be used with Code Fragment Virtual Fields.

Integers must not be used with Substring, Concatenation or Date Virtual Fields.

Predetermined Integers may be used for numeric predetermined joined fields.

Join Fields

RDML Fields of type

Commands **Integer**

may be used as a command parameter anywhere Numeric fields may be used, except where digits after the decimal point are required/expected.

Built-In Functions When used in Built-In Functions, integers are valid for arguments of type 'N'.

Special Values *NULL, *NAVAIL, *HIVAL, *LOVAL, *REMEMBERED_VALUE

Conversion It is wrong to convert Integers to/from Date and Time fields. Refer to [Field Type Conversions](#).

Usage Notes

- Working fields may be defined as TYPE(*INT) in functions or components. If no length is specified, the default of 4 is assumed. The default value will be *ZERO.
- When used in Built-In Functions, integers are valid for arguments of type 'N'. Fields of type Integer may only be used as numeric arguments or return values under the following conditions:
 - The minimum decimals for the argument or return value is 0.
 - The minimum length for the argument or return value is less than or equal to the implied length of the Integer field. For example, if the minimum length for the argument is 4, an Integer of 1 byte may not be used (as it only has an implied length of 3).
 - The maximum length for the argument or return value is 2147483647 OR the maximum length for the argument or return value is greater than or equal to the implied length of the Integer field. For example, if the maximum length for an argument is 4, an Integer of 2, 4, or 8 bytes may not be used (as they have implied lengths of 5 or higher).

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Also See

[1.1.7 Packed](#)

[1.1.8 Signed](#)

[1.1.10 Float](#)

[↑ 1.1 Field Types](#)

1.1.10 Float

A Float is an approximate numeric field that stores floating point numeric data (as opposed to fixed point like Signed and Packed). Floating point data is approximate. Not all values in the field type range can be precisely represented. Floats are classed as Numbers in LANSAs.

Fields of type **Float** may only be used in an arithmetic expression. Floats cannot be displayed in a component or function.

A Float is defined by the number of bytes used to store the value. The higher number of bytes, the more accurate the number. A 4 byte Float is accurate while the number of digits is less than or equal to 6. An 8 byte Float is accurate while the number of digits is less than or equal to 15.

The following table provides the accurate length for each of the possible byte lengths for a Float. The accurate length may be considered equivalent to the actual length of a signed or packed field. The table also notes the possible number of decimal places at runtime.

# Bytes	Accurate # digits (accurate length)	Possible decimal places
4	6	0 - 6
8	15	0 - 15

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Float field in the repository are:

Length	Floats may be 4 or 8 bytes in length. Decimals must always be zero.
Valid Keyboard Shift	Blanks
Allowed Attributes	AB, ASQN, CS, FE, ND, RA, RB, RL, RLTB.
Edit Mask	Not allowed.

Default *NULL

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files Floats may only be used in RDMLX Files. Floats may be used as real fields. Use of Floats as key fields is not recommended.

Logical Views Use of Floats as key fields is not recommended.

Virtual Fields Floats may only used with Mathematical Virtual Fields. Floats may be used as the source field(s).

Floats may be used with Code Fragment Virtual Fields. Floats must not be used with Substring, Concatenation or Date Virtual Fields.

Predetermined Fields Floats may be used for numeric predetermined joined fields.

Join Fields

RDML Fields of type

Commands

Float

may only be used in an arithmetic expression.

For example, DISPLAY, REQUEST, POP_UP, DEF_HEAD, DEF_FOOT DEF_BREAK, and DEF_LINE cannot support fields of type Float. Visual LANSA components have no mechanism for displaying fields of type Float.

Built-In Functions When used in Built-In Functions, floats are valid for arguments of type 'N'.

As the value for a field of type Float may have anywhere between 0 and 15 decimal places at execution time, it is generally not considered suitable as a numeric argument to a BIF as it is not possible to predict the actual number of decimal places.

Special Values *NULL, *NAVAIL, *HIVAL, *LOVAL, *REMEMBERED_VALUE

Conversion It is wrong to convert Floats to/from Date and Time fields. Refer to [Field Type Conversions](#).

Usage Notes

- Working fields may be defined as TYPE(*FLOAT) in functions or components. If no length is specified, the default of 8 is assumed.
- Fields of type **Float** may only be used in an arithmetic expression. Use of a Float as a command parameter that is not an arithmetic expression will result in a FFC Error. For example, a Float cannot be used in a BEGIN_LOOP command.
- Floats are generally not considered suitable as a numeric argument to a BIF.
- Exact comparisons using Floats is not recommended due to inaccuracy of the type. For example, the value stored in the field or saved in the table may not be exactly what was assigned to the field. However, comparison to *ZERO or *SQLNULL is fine.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- Floats are generally used in scientific or engineering applications and are not generally used in business applications.
- Floats are primarily supported in LANSA for use with Other Files and are not recommended for use in business applications.

Also See

[1.1.7 Packed](#)

[1.1.8 Signed](#)

[1.1.9 Integer](#)

[↑ 1.1 Field Types](#)

1.1.11 Date

Date is a fixed-length field with a length of 10, containing a date in ISO format: YYYY-MM-DD. The *NULL value is **1900-01-01**, as this value is valid across all supported databases.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Date field in the repository are:

Length	Dates must be 10 in length. Dates have no decimals.
Valid Keyboard Shift	Blanks
Allowed Attributes	AB, ASQN, CYDC, CYDU, CYDX, CS, FE, ISO, ND, RA, RL, RLTB. Note: ISO for display format must be selected.
Edit Mask	Not allowed.
Default	*SQLNULL. ASQN will be enabled by default.

Field Definition Notes

- By default, Dates are ISO format.

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files Dates may only be used in RDMLX Files. Dates may be used as real fields or keys. If used as key fields, take note of the

Warning
below.

Logical Views Dates may be used as keys to logical views.

Virtual Fields Dates may be used with Date Virtual Fields as the source field.
Dates may be used with Code Fragment Virtual Fields.

Predetermined Not allowed.

Join Fields

RDML Commands	Dates are classified as their own types and are not valid for numeric or alpha command parameters in RDML commands.
Built-In Functions	When used in Built-In Functions, Dates are classified as their own type (D) and are not valid for numeric or alpha arguments.
Special Values	*NULL, *HIVAL, *LOVAL, *REMEMBERED_VALUE
Conversion	Date fields may be converted to alpha, signed, packed, string or char. Refer to Field Type Conversions .

Usage Notes

- Working fields may be defined as TYPE(*DATE). Date literals are always specified in ISO format, for example: 2003-03-31.
- Dates can be used in Datecheck validation rules.

Warnings

- If this field is used as a key field, remove the *SQLNULL and ASQN attributes.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- To use a Date field, #MyDate, as an alpha argument, you simply use #MyDate.asString, explicitly stating that you want to use the string representation.

Also See

[1.1.13 DateTime](#)

[1.1.12 Time](#)

[↑ 1.1 Field Types](#)

1.1.12 Time

Time is a fixed-length field with a length of 8, containing a time in ISO format: HH:MM:SS. The *NULL value is **00:00:00** (i.e. midnight).

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining an Integer in the repository are:

Length	Times must be 8 in length. Times have no decimals.
Valid Keyboard Shift	Blanks
Allowed Attributes	AB, ASQN, CS, FE, ISO, ND, RA, RL, RLTB, TIMC, TIMU, TIMX. Note: ISO display format must be selected.
Edit Mask	Not allowed.
Default	*SQLNULL. ASQN will be enabled by default.

Field Definition Notes

- By default, Times are ISO format.

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files Times may only be used in RDMLX Files. Times may be used as real fields. Use of Times as key fields is not recommended. If used as key fields, take note of the

Warning
below.

Logical Views Use of Times as key fields is not recommended.

Virtual Fields Times may be used with Code Fragment Virtual Fields.

Predetermined Not allowed.

Join Fields

RDML Commands Times are classified as their own types and are not valid for numeric or alpha command parameters in RDML commands.

Built-In Functions	When used in Built-In Functions, Times are classified as their own types and are not valid for numeric or alpha arguments.
Special Values	*NULL, *NAVAIL, *REMEMBERED_VALUE
Conversion	Date fields may be converted to alpha, signed, packed, string or char. Refer to Field Type Conversions .

Usage Notes

- Working fields may be defined as TYPE(*TIME). Time literals are always specified in ISO format; for example: **22:58:35** or **03:08:05**. Note that seconds are optional (if omitted, they are assumed to be zero), so **22:58** or **03:08** would also be valid.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Warning

- If this field is used as a key field, remove the *SQLNULL and ASQN attributes.

Tips & Techniques

- To use a Time field, #MyTime, as an alpha argument, you simply use #MyTime.asString, explicitly stating that you want to use the string representation.

Also See

[1.1.13 DateTime](#)

[1.1.11 Date](#)

[↑ 1.1 Field Types](#)

1.1.13 DateTime

DateTime is a fixed-length field with a length of 19 (no fractional seconds) or between 21 and 29 (depending on number of positions after decimal point), containing a timestamp in ISO format: YYYY-MM-DD HH:MM:SS[.ffffff]. If no length is specified, the length will default to 26, which is the most portable maximum length and the ISO default: YYYY-MM-DD HH:MM:SS.ffffff.

Internally, a DateTime field always contains a DateTime in UTC (Universal Coordinated Time). UTC is the modern term for GMT (Greenwich Mean Time). LANSA automatically converts to and from UTC when required. The DUTC and SUTC attributes are used to define whether the DateTime is displayed and stored in the database in UTC or local time.

The *NULL value is **1900-01-01 00:00:00**.

DateTime literals may be specified with or without a time zone.

The literal format without a time zone is known as the ISO format (described above). In the ISO format, the DateTime literal is in UTC, so this specifies the **time in Greenwich, England, not the local time**. Both seconds and fractional seconds portions are optional (will be set to zero if not provided). Note that as this format contains a space, it must be enclosed in single quotes to allow the editor to correctly identify it as a DateTime literal. For example: '2004-02-03 00:10:30'

The literal format with a time zone is known as ISO 8601. In the ISO 8601 format, the time zone is always specified with the data, either as a Z, meaning UTC, or +/-hh:mm, being the difference from UTC. To distinguish ISO 8601 from the standard ISO format, as well as the time zone value, the blank between the date and the time is replaced with a T. Fractional seconds are optional (will be set to zero if not provided). For example: 1900-01-01T00:00:00Z is the *NULL value. Another example is 1994-11-05T08:15:30-05:00, which corresponds to November 5, 1994, 8:15:30 am, US Eastern Standard Time (the time zone is 5 hours behind UTC). 1994-11-05T13:15:30Z corresponds to the same instant.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining DateTime fields in the repository are:

Length May be 19 or 21 to 29 in length. Default is 26. Decimals are automatically calculated (if 19, Decimals are 0, otherwise Decimals

are Length - 20).

Valid Blanks
Keyboard
Shift

Allowed AB, ASQN, CS, DUTC, FE, ISO, ND, RA, RL, RLTB, SUTC,
Attributes TCYC, TCYU and TCYX.

Note: ISO display format must be selected.

Edit Not allowed.
Mask

Default *SQLNULL. ASQN will be enabled by default.

Field Definition Notes

- By default, DateTimes are ISO format.
- The DUTC (Display in UTC) attribute is disabled by default. If it is enabled, data is input and output in [UTC](#). If it is disabled, data is input and output in the local time zone. Generally, users prefer to see data in their own time zone, but it may be appropriate to display in UTC format for some cross-time zone reports, Web displays, etc.
 - DUTC affects the display and entry of DateTime data in reports, components (using PRIM_DTIM or PRIM_MCTL), web functions and web events. Note that if DUTC is disabled, the DateTime will be displayed and entered in the local time zone of the machine where the application code is being executed.
 - DUTC is irrelevant for field literals and intrinsic functions, where data is always manipulated in UTC.
 - DUTC is used for WAMS. The DateTime is always passed in XML in ISO 8601 UTC format, which is an XML standard. The std_datetime weblet takes care of visualization. The entered value is subsequently also submitted in ISO 8601 UTC format.
- The SUTC (Store in UTC) attribute is enabled by default. This means that when the field is on a file, and the file is written to or read from the database, the field will be stored in UTC rather than local time. This automatically allows applications executing anywhere in the world to process the DateTime with a known time zone. If using tools other than LANSAs to access the file, remember that the DateTime field is stored in UTC and not a local time zone.

- SUTC may be disabled if your data will only ever exist in one time zone, and all interfaces to the data will always be executed in the same time zone. If SUTC is disabled, the data will be saved in the local time zone of the machine where the database interface executes. For example, in a SuperServer application where file access is redirected to the server, data will be read and written in the local time zone of the server.
- SUTC is disabled by default for DateTime fields created by Load Other File, as most external applications store data in local time. If the external data is stored in UTC, you should enable SUTC.

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files DateTime fields may only be used in RDMLX Files. DateTimes may be used as real fields or keys. If used as key fields, take note of the

Warning
below.

Logical Views Use of DateTimes as key fields is not recommended.

Virtual Fields DateTimes may be used with Code Fragment Virtual Fields.

Predetermined Not allowed.

Join Fields

RDML Commands DateTimes are classified as their own types and are not valid for numeric or alpha command parameters in RDML commands.

Built-In Functions When used in Built-In Functions, Times are classified as their own types and are not valid for numeric or alpha arguments.

Special Values *NULL, *HIVAL, *LOVAL, *REMEMBERED_VALUE

Conversion Date fields may be converted to alpha, signed, packed, string or char. Refer to [Field Type Conversions](#).

Usage Notes

- Working fields may be defined as TYPE(*DateTime).
- If the DateTime field is not large enough for the fractional seconds provided,

then it will be rounded.

- Application code, including intrinsic functions, always processes DateTime in UTC format.

Warning

- If this field is used as a key field, remove the *SQLNULL and ASQN attributes.
- System Variables like *DATETIME provide values in local time. This is not consistent with the internal storage of DateTime fields which always contains a DateTime in UTC format. Instead, use the Now intrinsic.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- When you first enable your partition for RDMLX, new DateTime fields have SUTC on and DUTC off. This automatically allows multi-time zone applications, with data displayed and entered in local time, but saved in the database in UTC. This may initially be confusing, but in the end is what most applications need. If you wish your application to run all in UTC or all in local time, simply modify the default field attributes for the DateTime field type.
- To use a DateTime field, #MyDateTim, as an alpha argument, you simply use #MyDateTim.asString, explicitly stating that you want to use the string representation.
- To initialise a DateTime field to the current time or to set its default to the current time use the Now intrinsic. This ensures that the field is correctly set to UTC format.

Also See

[1.1.11 Date](#)

[1.1.12 Time](#)

[↑ 1.1 Field Types](#)

1.1.14 Binary

Binary is a fixed-length binary field, with a length between 1 and 32,767. Binary fields with lengths of 256 or less are equivalent to the existing Alpha field with the SBIN attribute enabled.

Binary fields are used to store binary data. This data is not subject to codepage conversion.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Binary field in the repository are:

Length	Binary fields may be 1 to 32,767 in length. Binary fields have no decimals.
Valid Keyboard Shift	Blanks
Allowed Attributes	AB, ASQN, CS, FE, ND, RA, RL, RLTB.
Edit Mask	Not allowed.
Default	*SQLNULL. ASQN will be enabled by default.

Field Definition Notes

- There is NO difference between Binary and VarBinary except how they are stored in the database.

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files Binary fields may only be used in RDMLX Files. Binary fields may be used as real fields. Binary fields may not be used as key fields.

Logical Views Binary fields may not be used as key fields.

Virtual Fields Binary fields may be used with Code Fragment Virtual Fields.

Predetermined Join Fields Binary fields may be used for lookup predetermined joined fields.

RDML Commands	DEF_HEAD, DEF_FOOT, DEF_BREAK, and DEF_LINE will not support printing of fields of type Binary. A FFC error will be generated if these Binary fields are used. Binary fields are manipulated as if they are Char fields except where sensitivity to their Binary characteristics will be incorporated.
Built-In Functions	When used in Built-In Functions, Binary fields are classified as their own types and are not valid for numeric or alpha arguments.
Special Values	*NULL, *NAVAIL, *REMEMBERED_VALUE
Conversion	Binary fields may be converted to character type fields. Refer to Field Type Conversions .

Usage Notes

- Working fields may be defined as TYPE(*BIN).
- If an SQL WHERE clause will/may be generated by a condition, these field types may only be compared to *NULL or *SQLNULL; any other comparison will be rejected.

Warnings

- Binary is treated as a fixed length type by some databases. Thus they pad the unused length and the padding character also differs between different databases. Other databases may treat it as variable length and thus not pad it at all. If data is then compared before and after, the result of the comparison will differ depending on the database that is being used. VarBinary is never padded and thus always compares correctly. Recommendation is to use VarBinary unless the binary data is always the same length

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Also See

[1.1.15 VarBinary](#)

[↑ 1.1 Field Types](#)

1.1.15 VarBinary

VarBinary is a variable-length binary field, with a maximum length between 1 and 32,767.

VarBinary fields are used to store binary data. This data is not subject to codepage conversion.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a VarBinary field in the repository are:

Length	VarBinary fields may be 1 to 32,767 in length. VarBinary fields have no decimals.
Valid Keyboard Shift	Blanks
Allowed Attributes	AB, ASQN, CS, FE, ND, RA, RL, RLTB.
Edit Mask	Not allowed.
Default	*SQLNULL. ASQN will be enabled by default.

Field Definition Notes

- There is NO difference between Binary and VarBinary except how they are stored in the database.

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files VarBinary fields may only be used in RDMLX Files. VarBinary fields may be used as real fields. VarBinary fields may not be used as key fields.

Logical Views VarBinary fields may not be used as key fields.

Virtual Fields VarBinary fields may be used with Code Fragment Virtual Fields.

Predetermined Join Fields VarBinary fields may be used for lookup predetermined joined fields.

RDML Commands	DEF_HEAD, DEF_FOOT, DEF_BREAK, and DEF_LINE will not support printing of fields of type VarBinary. A FFC error will be generated if these VarBinary fields are used. VarBinary fields are manipulated as if they are String fields except where sensitivity to their Binary characteristics will be incorporated.
Built-In Functions	When used in Built-In Functions, VarBinary fields are classified as their own types and are not valid for numeric or alpha arguments.
Special Values	*NULL, *NAVAIL, *REMEMBERED_VALUE
Conversion	VarBinary fields may be converted to character type fields. Refer to Field Type Conversions .

Usage Notes

- There is no working field type for VarBinary fields.
- If an SQL WHERE clause will/may be generated by a condition, these field types may only be compared to *NULL or *SQLNULL; any other comparison will be rejected.

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Also See

[1.1.14 Binary](#)

[↑ 1.1 Field Types](#)

1.1.16 DBCS Graphic String

A DBCS Graphic String is for use to store Japanese DBCS characters in Other files on an IBM i system. It is a type of graphic string with CCSID 300 encoding.

This type of field can only be created by Visual LANSA when loading an IBM i Other File into the repository. The IBM i Other File field is created as a Char/String type with pre-set Edit Code 'J' and a special Input Attribute 'SGRA'. It is invalid for use in any Visual LANSA File.

Field Definition Rules

Length	Total number of bytes of graphic characters plus two bytes for the <shift-out>/<shift-in> characters. Each graphic character is two bytes.
Valid Keyboard Shift	J
Allowed Attributes	SGRA, CS, FE.
Edit Mask	Not allowed.
Default	*NULL / *SQLNULL

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files IBM i Other File

Logical Views Not allowed.

Virtual Fields Not allowed.

Predetermined Not allowed.

Join Fields

RDML Commands DEF_HEAD, DEF_FOOT, DEF_BREAK, and DEF_LINE only support printing of RDML fields. Therefore, Char fields need to be converted to Alpha to be used in reports.

Built-In When used in Built-In Functions, Chars are valid for arguments

Functions of type 'A'.

Special Values *NULL, *NAVAIL, *REMEMBERED_VALUE

Conversion Refer to [Field Type Conversions](#)

Usage Notes

- Unsupported if defined as working fields.
- Unsupported if used as Visual LANSA File Field.

[↑ 1.1 Field Types](#)

1.1.17 BLOB

BLOB is a variable-length binary field of undefined maximum length.

The most common operation with BLOBs are saving files into the database and retrieving them so they can be viewed/edited/etc. In RDML and RDMLX, BLOB fields are manipulated as filenames.

Following is an example of saving a JPG into a BLOB:

```
#MYBLOB := 'C:\temp\mypicture.jpg'  
UPDATE FIELDS(#MYBLOB) IN_FILE(FILE1)
```

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a BLOB in the repository are:

Length	Length cannot be specified. No decimals.
Valid Keyboard Shift	Blanks
Allowed Attributes	AB, ASQN, CS, FE, LC, ND, RA, RL, RLTB. Note: LC and ASQN must always be defined and cannot be removed.
Edit Mask	Not allowed.
Default	*SQLNULL. ASQN will be enabled by default.

Field Definition Notes

- There is no working field type for BLOBs.

Usage Rules

Partition Type [RDMLX Enabled Partition](#)

Files BLOBS may only be used in RDMLX Files. BLOB fields may be used as real fields. BLOBS must not be used as key fields.

Logical Views BLOBS may not be used as key fields.

Virtual Fields Not applicable.

Predetermined Not applicable.

Join Fields

RDML Commands	If a BLOB or CLOB field is used, keep in mind that the field contains a filename, not the actual data in the object. In RDML and RDMLX, LANSALOB fields will be manipulated as filenames. It is only in database IO commands that the BLOB or CLOB actual data itself is handled by reading from or writing to the named file.
Built-In Functions	When used in Built-In Functions, BLOBs are classified as their own types and are not valid for numeric or alpha arguments.
Special Values	*SQLNULL, *NAVAIL, *REMEMBERED_VALUE, *EMPTY
Conversion	BLOBs are treated as file pointers and do not support conversion to a Numeric or Date/Time field type. Refer to Field Type Conversions .

Usage Notes

- BLOBs cannot be part of any key (e.g. for Access Routes, etc.)
- You cannot display or print BLOB data.
- If a BLOB field is added to the list, keep in mind that the field contains a filename, not the actual data in the object.
- If an SQL WHERE clause will/may be generated by a condition, these field types may only be compared to *NULL or *SQLNULL; any other comparison will be rejected.
- Rather than the default property **.Value**, fields of type BLOB have a default property called **.FileName** to clearly indicate that changing the "value" of the field is actually changing its default property which is a file name property.
- BLOB fields are subject to certain restrictions:
 - They cannot be used in SELECT_SQL commands.
 - They cannot be used in a condition; the exception is comparison against *NULL, or *SQLNULL.
 - Changes may not be logged (and therefore rollback may have no effect) on some or all DBMSs.
 - The attribute LC is always enabled, and cannot be disabled. This will affect filenames initially. BLOB, as a binary type, allows any type of data so lowercase is meaningless within the file.
 - The attribute ASQN is always enabled, and cannot be disabled.

- When BLOB and CLOB data is read from the database, files are automatically created in the directory structures under the LPTH= directory (for information, refer to [Standard X_RUN Parameters](#)).

Platform Considerations

- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- The recommended design when using BLOB and CLOB fields is to put them in a separate file from the rest of the fields using the same key as the main file. This forces programmers to do separate IOs to access the BLOB and CLOB data, thus reducing impact on database performance from indiscriminate use of this data. It is also the most portable design ensuring that the non-BLOB and non-CLOB data can be quickly accessed at all times.
- The LOB directory files created on read are occasionally not deleted at the end of your LANSAsession. A special process, *LOBCLNUP, can be executed occasionally to cleanup the LOB directory structure. The process needs to be run by a user with sufficient authority to remove files that may have been created by other users. For example, on IBM i, use the following command:
LANSAREQUEST(X_RUN)
X_RUNADPRM('PROC=*LOBCLNUP')

Also See

[1.1.18 CLOB](#)

[↑ 1.1 Field Types](#)

1.1.18 CLOB

CLOB is a variable-length character field of undefined maximum length. CLOBs can be used for saving files into the database and retrieving them so they can be viewed/edited/etc. In RDML and RDMLX, CLOB fields are manipulated as filenames.

Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a CLOB field in the repository are:

Length	Length cannot be specified. No decimals.
Valid Keyboard Shift	Blanks, O, E, J or W.
Allowed Attributes	AB, ASQN, CS, FE, LC, ND, RA, RL, RLTB, SUNI. Note: LC and ASQN must always be defined and cannot be removed.
Edit Mask	Not allowed.
Default	*SQLNULL. ASQN will be enabled by default.

Field Definition Notes

- There is no working field type for CLOBs.

Usage Rules

Partition Type	RDMLX Enabled Partition
Files	CLOBs may only be used in RDMLX Files. CLOB fields may be used as real fields. CLOBs must not be used as key fields.
Logical Views	CLOBs must not be used as key fields.
Virtual Fields	Not applicable.
Predetermined	Not applicable.
Join Fields	
RDML Commands	If a BLOB or CLOB field is used, keep in mind that the field contains a filename, not the actual data in the object. In RDML and RDMLX, LANSALOB fields will be manipulated as

filenames.

Built-In Functions	When used in Built-In Functions, CLOBs are classified as their own types and are not valid for numeric or alpha arguments.
Special Values	*SQLNULL, *NAVAIL, *REMEMBERED_VALUE, *EMPTY
Conversion	CLOBs are treated as file pointers and do not support conversion to a Numeric or Date/Time field type. Refer to Field Type Conversions .

Usage Notes

- There are no working field types for CLOBs.

CLOBs cannot be part of any key (e.g. for Access Routes, etc.)

- You cannot display or print CLOB data.
- If a CLOB field is added to the list, keep in mind that the field contains a filename, not the actual data in the object.
- If an SQL WHERE clause will/may be generated by a condition, these field types may only be compared to *NULL or *SQLNULL; any other comparison will be rejected.
- Rather than the default property **.Value**, fields of type CLOB have a default property called **.FileName** to clearly indicate that changing the "value" of the field is actually changing its default property which is a file name property.
- CLOB fields are subject to certain restrictions:
 - They cannot be used in SELECT_SQL commands.
 - They cannot be used in a condition; the exception is comparison against *NULL, or *SQLNULL.
 - Changes may not be logged (and therefore rollback may have no effect) on some or all DBMSs.
 - The attribute LC is always enabled, and cannot be disabled. This will affect filenames initially, but could also eventually affect the content for CLOBs.
 - The attribute ASQN is always enabled, and cannot be disabled.
- When BLOB and CLOB data is read from the database, files are automatically created in the directory structures under the LPTH= directory (for information, refer to [Standard X_RUN Parameters](#)).

Platform Considerations

- When reading CLOB fields on IBM i, the file created on the IFS will have the same CCSID as the native CLOB or DBCLOB column on the database table; no data conversion is performed on the data. That includes DBCLOB columns with CCSID 1200 or 13488.
- Refer to Platform Considerations in [1.1.1 Field Type Considerations](#).

Tips & Techniques

- The recommended design when using BLOB and CLOB fields is to put them in a separate file from the rest of the fields using the same key as the main file. This forces programmers to do separate IOs to access the BLOB and CLOB data, thus reducing impact on database performance from indiscriminate use of this data. It is also the most portable design ensuring that the non-BLOB and non-CLOB data can be quickly accessed at all times.
- The LOB directory files created on read are occasionally not deleted at the end of your LANSAsession. A special process, *LOBCLNUP, can be executed occasionally to cleanup the LOB directory structure. The process needs to be run by a user with sufficient authority to remove files that may have been created by other users. For example, on IBM i, use the following command: LANSAREQUEST(X_RUN)
X_RUNADPRM('PROC=*LOBCLNUP')

Also See

[1.1.17 BLOB](#)

[↑ 1.1 Field Types](#)

1.1.19 Boolean

Boolean fields have only two possible values: either False (0) or True (1). Please review the general [1.1.1 Field Type Considerations](#).

Field Definition Rules

Rules for defining a Boolean field in the repository are:

Length	Length cannot be specified. No decimals.
Valid Keyboard Shift Blanks	
Allowed Attributes	ASQN
Edit Mask	Not allowed.
Default	*NULL (False)

Field Definition Notes

- None.

Usage Rules

Partition Type	RDMLX Enabled Partitions
Files	Booleans may only be used in RDMLX Files. Booleans may be used as real or key fields.
Logical Views	Booleans may be used as keys to logical views. Booleans may be used in Select/Omit criteria with COMP() EQ/NE 'True' or 'False'
Virtual Fields	Booleans may be used as virtual field; they may only be assigned a value using Code Fragments.
Predetermined Join Fields	Booleans may be used for lookup predetermined joined fields.
RDML Commands	Booleans are classified as their own type and are not valid for numeric or alpha command parameters in RDML commands.
Built-In Functions	Booleans are classified as their own type and are not valid for numeric or alpha command parameters in RDML commands.
Special	*NULL, *NAVAIL, *HIVAL, *LOVAL,

Values	*REMEMBERED_VALUE
Conversion	Booleans may be converted to alpha, integer, signed, packed, string or char. In numeric conversions, False becomes 0, and True becomes 1. In Character/String conversions, the target is populated with the word "False" or "True". Refer to Field Type Conversions.

Usage Notes

- Working fields may be defined as TYPE(*BOOLEAN).
- In a SELECT_SQL Where clause, you must use 0 and 1, the keywords True and False are not supported.
- In Select/Omit criteria, Boolean literals must be specified in capitals and surrounded by quotes. I.e. 'TRUE' or 'FALSE'.

Platform Considerations

- Refer to **Platform Considerations** in [1.1.1 Field Type Considerations](#).

Also See

[↑ 1.1 Field Types](#)

1.2 Field Definitions

1.2.20 Alias Name	1.2.16 Field Attributes	1.2.3 Field Type
1.2.11 Allocated Length (IBM i only)	1.2.10 Field Column Heading	1.2.17 Input Attributes
1.2.5 Decimals	1.2.8 Field Description	1.2.13 Keyboard Shift
1.2.6 Default Value	1.2.2 Field Identifier	1.2.18 Output Attributes
1.2.12 Edit Mask	1.2.9 Field Label	1.2.19 Prompting
1.2.14 Enable Field for RDMLX	1.2.4 Field Length	1.2.7 Reference Field
	1.2.1 Field Name	1.2.15 System Field

Also See

[1.1 Field Types](#)

[1.3 Field Visualizations](#)

[1.4 Field Rules and Triggers](#)

[1.5 Field Help Text](#)

[Fields](#) in the *User Guide*

[Creating Fields](#) in the *Developer Guide*

[↑ 1. Fields](#)

1.2.1 Field Name

Mandatory.

Specify the name of the field to be created in the LANSAs Repository.

Refer to [LANSAs object name](#).

Rules

- Refer to [LANSAs object name](#).

Warnings

- Refer to [LANSAs object name](#).

Tips & Techniques

- Refer to [LANSAs object name](#).

Platform Considerations

- Refer to [LANSAs object name](#).

[Also See](#)

[1.2.2 Field Identifier](#)

[↑ 1.2 Field Definitions](#)

1.2.2 Field Identifier

Mandatory.

Specify the identifier of the field to be stored in the LANSAs Repository. Field identifiers are not case sensitive. By default, field identifiers are often converted to upper case characters in LANSAs.

Rules

- Must be a valid [LANSAs object name](#).
- Cannot be the same as the alias name or the alias for another field.

Warnings

- Avoid the use of field identifiers like SQLxxx, as this may cause problems when used in functions that use SQL (Structured Query Language) facilities. (i.e. Command SELECT_SQL.)

Platform Considerations

- IBM i: Use of identifiers of more than 6 characters in length is not recommended if you are writing your own RPG application programs. Refer to the *RPG Programmer's Reference Guide*.

Tips & Techniques

- Use alias name for COBOL or PL/1 long names.
- Field naming standards will assist in the maintenance of your LANSAs applications. Refer to [Object Naming Standards](#) in the *LANSAs Application Design Guide*.
- LANSAs recommends a corporate data dictionary approach for creating fields in the LANSAs Repository. Refer to [Corporate Data Dictionary Concepts](#) in the *Developer Guide*.

Also See

[1.2.1 Field Name](#)

[↑ 1.2 Field Definitions](#)

1.2.3 Field Type

Mandatory.

Specify the type of field to be created in the LANSAs Repository.

Allowable [1.1 Field Types](#) are defined at the partition level. Refer to [1.1.1 Field Type Considerations](#).

It is important to set the type of the field before specifying other attributes because attributes are dependent on the field type. (Refer to [1.2.17 Input Attributes](#) and [1.2.18 Output Attributes](#).)

Rules

- Rules are specific to the [1.1 Field Types](#).
- Cannot be entered if a [1.2.7 Reference Field](#) has been specified.

Warnings

- Refer to [1.1 Field Types](#).

Tips & Techniques

- Refer to [1.1 Field Types](#).

Platform Considerations

- Refer to [1.1 Field Types](#).

Also See

[1.2.4 Field Length](#)

[1.2.5 Decimals](#)

[↑ 1.2 Field Definitions](#)

1.2.4 Field Length

Mandatory.

Specify the length of the field to be created in the LANSAs Repository. Field length is dependent upon the [1.1 Field Types](#).

Rules

- Refer to [1.1 Field Types](#).
- Cannot be entered if a [1.2.7 Reference Field](#) has been specified.

Warnings

- Refer to [1.1 Field Types](#).

Tips & Techniques

- Refer to [1.1 Field Types](#).

Platform Considerations

- Refer to [1.1 Field Types](#).

Also See

[1.2.5 Decimals](#)

[↑ 1.2 Field Definitions](#)

1.2.5 Decimals

Specify the number of decimals for fields of numeric type. The specification of decimals is dependent upon the [1.1 Field Types](#).

Rules

- Refer to [1.1 Field Types](#).
- Must be less than or equal to [1.2.4 Field Length](#) (total number of digits).
- Cannot be entered if a [1.2.7 Reference Field](#) has been specified.

Warnings

- Refer to [1.1 Field Types](#).

Tips & Techniques

- Refer to [1.1 Field Types](#).

Platform Considerations

- Refer to [1.1 Field Types](#).

Also See

[1.2.4 Field Length](#)

[↑ 1.2 Field Definitions](#)

1.2.6 Default Value

Specify the default value for a field. This value is used as defaults on screens, reports and fields in a function.

If no default value is specified, the field defaults to *BLANKS (blanks) for character fields, *ZERO (zero) for numeric fields and *NULL (null value) for other fields.

Rules

Allowable values include:

- system variable such as *BLANKS, *ZERO, *DATE, *NULL or any other variables specifically defined at your installation.
- alpha literal (in quotes) such as 'NSW', 'BALMAIN' or 'Australia'.
- numeric literal such as 1, 10.43, -1.341217

Additional rules:

- Refer to [1.1 Field Types](#).
- Cannot be entered if a [1.2.7 Reference Field](#) has been specified.

Warnings

- Refer to [1.1 Field Types](#).

Tips & Techniques

- Refer to [1.1 Field Types](#).

Platform Considerations

- Refer to [1.1 Field Types](#).

Also See

[General Variables](#)

[↑ 1.2 Field Definitions](#)

1.2.7 Reference Field

Specify the name of the field which should be "referred" to. The new field will inherit the following characteristics from the reference field entered:

- Type
- Length
- Number of decimal positions
- Default value
- Edit mask
- Keyboard Shift
- Input attributes and Output attributes.

You cannot change these characteristics while the reference field is specified, and they are automatically updated if the reference field is changed. For example, if the length of the reference field is changed, the same changes will automatically be made to the fields that refer to it.

Initially, the prompt process and function are inherited, but you can change these details if required.

Rules

- The reference field must already exist in the repository.

Warnings

- In an RDML partition, if the flag *IMPREFFLDNOPROP is in the system data area DC@OSVEROP, the input and output attributes will not be updated during an import or export. If the reference field is subsequently changed, the changes will be propagated to the fields referencing the reference field. For further information refer to *Reference field propagation in import* [Export and Import settings](#) of the *LANSA for i User Guide*.
- In an RDMLX partition, the attributes of a field that is defined by a reference field cannot be changed. The *IMPREFFLDNOPROP flag in the system definitions is ignored.

Tips & Techniques

- All LANSA modeling tools that use a data type approach, will use reference fields when building field definitions.
- Make each referred field a [1.2.15 System Field](#) so that it won't be deleted.

Also See

[1.1 Field Types](#)

[↑ 1.2 Field Definitions](#)

1.2.8 Field Description

Mandatory.

Specify the description associated with the field. The field description text will be used as the default field description on screens and reports generated by LANSAs.

Rules

- A field description must be entered for each language defined for the partition.

Platform Considerations

- IBM i: Use of CUA standards for identification are recommended.

Also See

[1.2.9 Field Label](#)

[1.2.10 Field Column Heading](#)

[↑ 1.2 Field Definitions](#)

1.2.9 Field Label

Mandatory.

Specify the label associated with the field stored. The field label text can be used as the field description when a field is used on screens and reports generated by LANSAs.

Rules

- Maximum length is 15.
- A field label must be entered for each language defined for the partition.

Platform Considerations

- IBM i: Use of CUA standards for identification are recommended.

Also See

[1.2.8 Field Description](#)

[1.2.10 Field Column Heading](#)

[↑ 1.2 Field Definitions](#)

1.2.10 Field Column Heading

Mandatory.

Specify the column heading associated with the field. The field column heading text will be used as the default column headings text when a field is used in a list on a screen or report generated by LANSAs.

Rules

- A field column heading must be entered for each language defined for the partition.

Tips & Techniques

- Match column heading text length to the field length. For example, a 2 character state code might have a 2 character column heading of ST instead of using STATE CODE.
- Single line column headings require less display area on screens and reports.

Platform Considerations

- IBM i: Use of CUA standards for identification are recommended.

Also See

[1.2.8 Field Description](#)

[1.2.9 Field Label](#)

[↑ 1.2 Field Definitions](#)

1.2.11 Allocated Length (IBM i only)

What is it for?

The Allocated length is only used when creating files on IBM i. It defines the space to be reserved for variable length columns in each row. Column values with lengths less than or equal to the allocated value are stored in the fixed-length portion of the row. Column values with lengths greater than the allocated value are stored in the variable-length portion of the row and require additional input/output operations to retrieve.

Rules:

The allocated length cannot exceed the [1.2.4 Field Length](#).

Allocated length is only available with these field types: String, NVarchar, VarBinary, CLOB & BLOB.

If the field has keyboard shift E, O, or J, the allocated field length must be 0, or greater than 4.

If the field has the attribute SUNI, the allocated field length is specified in characters, not bytes.

For example:

If you enter an allocated length of 32000, and the file is created via DDS, VARLEN(32000) will be specified for the field definition.

If the file is created via SQL's CREATE TABLE statement, ALLOCATE(32000) will be specified for the column definition.

For further details, please refer to the *IBM DDS Reference* and *SQL Reference* manuals.

[↑ 1.2 Field Definitions](#)

1.2.12 Edit Mask

Edit Mask allows you to specify the format in which you want your data returned for the application.

Edit Code

Specify the letter or number representing the format for output-capable numeric fields. This indicates the formatting to be done before a field is displayed or printed. The following editing can be done, depending on which edit code is specified:

- Leading zeros can be suppressed.
- The field can be punctuated with commas and decimal points to show decimal position and to group digits by threes.
- Negative values can be displayed with a minus sign or CR to the right. Zero values can be displayed as zero or blanks.

Refer to [Standard Field Edit Codes](#) for edit codes that are supported in this version of LANSAs.

Edit Word

If you cannot accomplish the desired editing by using a predefined edit code, you may specify an edit word instead. An edit word specifies the form in which the field values are to print and clarifies the data by inserting characters, such as decimal points, commas, floating- and fixed-currency symbols, and credit balance indicators. Also use it to suppress leading zeros and to provide asterisk fill protection.

Warnings

- Use of edit words should only be attempted by experienced users as the validity checking done by LANSAs is unsophisticated. Invalid edit words may pass undetected into the system and cause subsequent failures when attempting to create database files or compile programs. Refer to IBM manual Data Description Specifications for more details about EDTCDE (Edit Code) and EDTWRD (Edit Word) Keywords.

Rules

- Edit codes cannot be used with all numeric field types. Refer to [1.1 Field Types](#).
- Edit Codes cannot be entered if a [1.2.7 Reference Field](#) has been specified as the code is inherited from the referenced field.

Tips & Techniques

- Use of edit masks for packed and signed fields is strongly recommended.
- Note: Edit word processing involving floating currency symbols is handled differently by the operating system for screens and reports. If such a problem occurs, it is best overcome by the use of a virtual field for report production and only using the real field for screen display.

Also See

[1.1.1 Field Type Considerations](#)

[↑ 1.2 Field Definitions](#)

1.2.13 Keyboard Shift

Specify the keyboard shift to limit what the user can type into a field.

Note - V11 SP5 onwards:

- Keyboard Shift U is no longer an option. New Field Attribute SUNI indicates if a field is a Unicode field or not.
- Keyboard shift settings can be changed and maintain via the LANSAs Editor.

Valid keyboard shift values are based on [1.1 Field Types](#).

Rules

Allowable values are:

Keyboard shift	Meaning	Data Type Allowed
A	Alpha shift	Alpha
Blank		Alpha / Numeric
D	Digits only 0 - 9	Alpha/Numeric
I	Inhibit entry (no keyboard entry allowed)	Alpha/Numeric
M	Numeric keys	0 - 9, plus, minus, comma, dash, space, period
N	Numeric shift	Alpha / Numeric
S	Signed numeric	Numeric
W	Katakana (for Japan only)	Alpha
X	Alphabetic only	Alpha. A - Z, comma, period, dash, space
Y	Numeric only	Numeric

Additional rules are:

- Cannot be entered if a [1.2.7 Reference Field](#) has been specified.

Tips & Techniques

- When using a DBCS LANSA language on DBCS operating systems, the IME mode is set dependent on the type of keyboard shift to allow the construction of DBCS characters from phonetics.
- Keyboard Shift U is not valid for fields defined in functions. Therefore, any field definition commands that reference the original Other File field with Keyboard Shift U, must specify a valid value for the SHIFT parameter. Use SHIFT(*BLANKS) if users will only be entering SBCS data, otherwise set the SHIFT parameter to another appropriate value.

Platform Considerations

- IBM i: The IBM i DBCS or IGC data types J, E & O have been implemented as the Alpha LANSA data type with LANSA keyboard shift J, E & O respectively.

J Alpha, Used for DBCS only.

E Alpha. Used for all DBCS characters or all SBCS characters.
Both DBCS and SBCS are not allowed in the same field.

O Alpha. Used for mixed DBCS and SBCS

- IBM i: Refer to the IBM manual *Data Description Specifications* for more details about DDS Data Type/Keyboard Shift for display files (Position 35).

Also See

[1.1.1 Field Type Considerations](#)

1.2.14 Enable Field for RDMLX

Specify if this field is enabled for full RDMLX so that extended field definition characteristics can be used. Refer to [What Classifies a Field as RDMLX.](#))

This option is only available in an RDMLX Enabled Partition.

The default value for this option is controlled in the [RDMLX Partition Settings.](#)

Tips & Techniques

- Once enabled for RDMLX, you can change the field type to an RDMLX field type. Refer to [Field Type Considerations](#) for details about RDML and RDMLX fields.
- When you change the field option to RDMLX, the field definition must meet all RDMLX requirements before you can save it.
- It is recommended that you review the [RDML and RDMLX Partition Concepts](#) in the Administrator Guide.

Implications

- All editing must be performed using Visual LANSA. You cannot edit RDMLX Fields from LANSA for i.
- RDMLX fields cannot be used by RDML Objects. Once it becomes an RDMLX field, a field cannot be used within any RDML File, RDML Function or RDML Component. This change may have major implications to your existing applications. (RDML Fields can be used by RDMLX objects.)

Warning

- If a field is changed so it is no longer enabled for RDMLX, then all RDMLX features must be removed before it can be saved.

[↑ 1.2 Field Definitions](#)

1.2.15 System Field

Mandatory. Default=No (not selected).

Specify if this field is to be considered a LANSAsystem field. LANSAsystem fields cannot be deleted from the repository while it remains identified as a system field.

The standard shipped LANSAsystem fields are inserted into each new partition.

Tips & Techniques

- Make all reference fields system fields. Refer to [1.2.7 Reference Field](#).

[↑ 1.2 Field Definitions](#)

1.2.16 Field Attributes

Field attributes are used to control how a field is displayed when it is used as an input field ([1.2.17 Input Attributes](#)) and how a field is displayed when it is used for output ([1.2.18 Output Attributes](#)).

Field Attributes

Specify the field attributes. You may manually edit the list of attributes for a field, or the list will automatically be updated as you select the codes from the lists for specific Attribute Types. The allowable [1.2.17 Input Attributes](#) and [1.2.18 Output Attributes](#) will vary by [1.1 Field Types](#).

Attribute Type

Specify the desired attribute type from the drop down list. The allowable Attribute Types for the [1.2.17 Input Attributes](#) and [1.2.18 Output Attributes](#) will vary by [1.1 Field Types](#).

[↑ 1.2 Field Definitions](#)

1.2.17 Input Attributes

Input Field Attribute Types

Specify how a field is displayed when it is used as an input field in a function or a form.

Rules

Allowable values will vary by [1.1 Field Types](#). Groupings include:

- Common Attributes
- SAA/CUA Attributes - Functions Only
- Colors - Functions Only
- GUI Attributes - Function Only

Tips & Techniques

- Most input attributes, such as color, are used in functions because other LANSAs screens (forms, web functions, etc.) use other techniques such as visual style to define how fields are displayed.
- Input attributes such as Hidden Field (for passwords) or Lowercase (allow lower case text to be entered) can be used for all types of applications.

Input Attributes

Specify the field attribute to be displayed when it is used as an input field in a function or a form.

Rules

Common Attributes

Allowable values for Common Attributes are:

Attribute	Description / comments
AB	Allow blank.
ASQN	Allow *SQLNULL assignment.
BL	Display blinking.
CBOX	Check box.
CS	Display with column separators.
FE	Field Exit key required.

HI	Display with high intensity.
LC	Lowercase entry allowed. If you do NOT set this attribute, refer to PC Locale uppercasing requested in Review or Change a Partition's Multilingual Attributes in the <i>LANSA for i User Guide</i> .
ME	Mandatory entry check required.
MF	Mandatory fill check required.
ND	Non-display (hidden field).
RA	Auto record advance field.
RB	Right adjust and blank fill.
RL	Move cursor right to left.
RLTB	Tab cursor right/left top/bottom.
RZ	Right adjust and zero fill.
SBIN	Store in binary format. This is a special attribute provided for fields that need to contain imbedded packed or signed fields. Refer to Use of Hex Values, Attributes, Hidden/Imbedded Decimal Data in the <i>LANSA Application Design Guide</i> (Use of Hex Values, Attributes, Hidden/Imbedded Decimal Data).
SREV	Store in reverse format. This is a special attribute provided for bi-directional languages. Refer to The SREV Field Attribute in the <i>LANSA Multilingual Application Design Guide</i> .
SUNS	Store integer values as unsigned binary values.
SUNI	Store in Unicode. This is a special attribute which allows data from different languages to be stored in the database without data loss through code page conversion. For new fields, field type NChar or NVarChar is recommended, rather than adding the SUNI attribute.
VN	Valid name check required.

CUA Attributes - Function Only

Allowable values for SAA/CUA Attributes - Function Only are:

Attribute	Description / comments
ABCH	Action (menu) bar and pull-down choices
FKCH	Function key information
PBBR	Brackets
PBCE	Protected field (emphasized)
PBCH	Choices shown on menu
PBCM	Field column headings
PBCN	Protected field (normal)
PBEE	Input capable field (emphasized)
PBEN	Input capable field (normal)
PBET	Emphasized text
PBFP	Field prompt / label or description details
PBGH	Group headings
PBIN	Instructions to user
PBNT	Normal text
PBPI	Panel identifier
PBPT	Panel title
PBSC	Choice last selected from menu
PBSI	Scrolling information
PBSL	Separator line
PBUC	Choices that are not available
PBWB	Pop-up window border

Note that normally only PBEN and PBEE would be specified as input attributes.

Colors - Functions Only

Allowable values for Colors - Functions Only are:

Attribute	Description / comments
BLU	Display with color blue.
GRN	Display with color green.
PNK	Display with color pink.
RED	Display with color red.
TRQ	Display with color turquoise.
WHT	Display with color white.
YLW	Display with color yellow.

GUI Attributes – Functions Only

Allowable values for GUI Attributes – Functions Only are:

Attribute	Description / comments
CBOX	Present field value as a GUI WIMP Check Box.
DDxx	Drop Down Represents the field with the corresponding GUI WIMP construct.
PBnn	Push Button
RBnn	Radio Button

Tips & Techniques

- Only one color can be specified for a field.
- Use of colors may affect other attributes.

Platform Considerations

- IBM i: Refer to IBM manual Data Description Specifications for more details. Keywords that should be reviewed are CHECK, COLOR and

DSPATR.

Also see

[1.2.18 Output Attributes](#)

[↑ 1.2 Field Definitions](#)

1.2.18 Output Attributes

Output Field Attribute Types

Specify how a field is displayed when it is used for output in a function or a form.

Rules

Allowable values will vary by [1.1 Field Types](#). Grouping include:

- Common Attributes
- SAA/CUA Attributes - Functions Only
- Colors - Functions Only
- Record Stamping - Create
- Record Stamping - Create & Update
- Record Stamping - Update
- GUI Attributes - Functions Only
- UD Reporting Attributes - Functions Only

Output Attributes

Specify the field attribute to be displayed when it is used for output in a function or form

Rules

Common Attributes

Allowable values for Common Attributes are:

Attribute	Description / comments
ASQN	Allow *SQLNULL assignment.
BL	Display blinking.
CBOX	Present field value as a GUI WIMP Check Box.
CDTX	Stamped with the date (*SYSFMT8) that the record was created or last updated.
CS	Display with column separators.
CYDX	Stamped with the date (CCYYMMDD) that the record was created or last updated.

DATX	Stamped with the date (*SYSFMT) that the record was created or last updated.
DUTC	Display in <u>UTC</u> . DateTimes only.
FUNX	Stamped with the name of the LANSAs function or component that either created or last updated the record.
HI	Display in high intensity.
ISO	Display in ISO Format. Date, Time, and DateTime only.
JNMX	Stamped with the name of the job that either created or last updated the record.*
JNRX	Stamped with the number of the job that either created or last updated the record.*
ND	Non-display (hidden field).
PROX	Stamped with the name of the LANSAs Process that either created or last updated the record.
RA	Auto record advance field
SBIN	Store in binary format. This is a special attribute provided for fields that need to contain imbedded packed or signed fields. Refer to the LANSAs Application Design Guide (Use of Hex Values, Attributes, Hidden/Imbedded Decimal Data).
SREV	Store in reverse format. This is a special attribute provided for bidirectional languages. Refer to The SREV Field Attribute in the <i>LANSAs Multilingual Application Design Guide</i> .
SUNS	Store integer values as unsigned binary values
SUNI	Store in Unicode. This is a special attribute which allows data from different languages to be stored in the database without data loss through code page conversion. For new fields, field type NChar or NVarChar is recommended, rather than adding the SUNI attribute.
SUTC	Store in the database in <u>UTC</u> . Datetimes only.
TCDX	Stamped with the DateTime (HHMMSS+*SYSFMT8) that the record was created or last updated.

TDSX	Stamped with the DateTime (HHMMSS+*SYSFMT) that the record was created or last updated.
TIMX	Stamped with the time (HHMMSS) that the record was created or last updated.
TYDX	Stamped with the DateTime (HHMMSSYYMMDD) that the record was created or last updated.
Urxx	Associate field with a User Defined Reporting Attribute (URxx). Provides access to IBM i DDS statements for printer files.

SAA/CUA Attributes – Functions Only

Allowable values for SAA/CUA Attributes – Functions Only are:

Attribute	Description / comments
ABCH	Action (menu) bar and pull-down choices
FKCH	Function key information
PBBR	Brackets
PBCE	Protected field (emphasized)
PBCH	Choices shown on menu
PBCM	Field column headings
PBCN	Protected field (normal)
PBEE	Input capable field (emphasized)
PBEN	Input capable field (normal)
PBET	Emphasized text
PBFP	Field prompt / label or description details
PBGH	Group headings
PBIN	Instructions to user
PBNT	Normal text

PBPI	Panel identifier
PBPT	Panel title
PBSC	Choice last selected from menu
PBSI	Scrolling information
PBSL	Separator line
PBUC	Choices that are not available
PBWB	Pop-up window border

Colors – Functions Only

Allowable values for Colors – Functions Only are:

Attribute	Description / comments
BLU	Display with color blue.
GRN	Display with color green.
PNK	Display with color pink.
RED	Display with color red.
TRQ	Display with color turquoise.
WHT	Display with color white.
YLW	Display with color yellow.

Record Stamping – Create

Allowable values for Record Stamping – Create are:

Attribute	Description / comments
CDTC	Stamped with the date (*SYSFMT8) that the record was created.
CYDC	Stamped with the date (CCYYMMDD) that the record was created.

DATC	Stamped with the date (*SYSFMT) that the record was created.
FUNC	Stamped with the name of the LANSAs function or component that created the record.
JNMC	Stamped with the name of the job that created the record.*
JNRC	Stamped with the number of the job that created the record.*
PROC	Stamped with the name of the LANSAs Process that created the record.
TCDC	Stamped with the DateTime (HHMMSS+*SYSFMT8) that the record was created.
TCYC	Stamped with the DateTime (YYYY-MM-DD HH:MM:SS) that the record was created.
TDSC	Stamped with the DateTime (HHMMSS+*SYSFMT) that the record was created.
TIMC	Stamped with the time (HHMMSS) that the record was created.
TYDC	Stamped with the DateTime (HHMMSSYYMMDD) that the record was created.
USRC	Stamped with the name of the user that created the record.* If *LONG_USER_AUDIT is enabled then the user name will be the audit user (up to 256 characters) if the SET_SESSION_VALUE USER_AUDIT has been set, otherwise the current authenticated user name is used if available, otherwise the current user name is used. If *LONG_USER_AUDIT is not enabled then the user name will be the audit user (up to 10 characters) if the SET_SESSION_VALUE USER_AUDIT has been set, otherwise the current user name is used.
YMDC	Stamped with the date (YYMMDD) that the record was created.

Record Stamping – Create & Update

Allowable values for Create & Update are:

Attribute	Description / comments

CDTX	Stamped with the date (*SYSFMT8) that the record was created or last updated.
CYDX	Stamped with the date (CCYYMMDD) that the record was created or last updated.
DATX	Stamped with the date (*SYSFMT) that the record was created or last updated.
FUNX	Stamped with the name of the LANSAs function or component that either created or last updated the record.
JNMX	Stamped with the name of the job that either created or last updated the record.*
JNRX	Stamped with the number of the job that either created or last updated the record.*
PROX	Stamped with the name of the LANSAs Process that either created or last updated the record.
TCDX	Stamped with the DateTime (HHMMSS+*SYSFMT8) that the record was created or last updated.
TCYX	Stamped with the DateTime (YYYY-MM-DD HH:MM:SS) that the record was created or last updated.
TDSX	Stamped with the DateTime (HHMMSS+*SYSFMT) that the record was created or last updated.
TIMX	Stamped with the time (HHMMSS) that the record was created or last updated.
TYDX	Stamped with the DateTime (HHMMSSYYMMDD) that the record was created or last updated.
USRX	Stamped with the name of the user that either created or last updated the record.* If *LONG_USER_AUDIT is enabled then the user name will be the audit user (up to 256 characters) if the SET_SESSION_VALUE USER_AUDIT has been set, otherwise the current authenticated user name is used if available, otherwise the current user name is used. If *LONG_USER_AUDIT is not enabled then the user name will be the audit user (up to 10 characters) if the SET_SESSION_VALUE

	USER_AUDIT has been set, otherwise the current user name is used.
YMDX	Stamped with the date (YYMMDD) that the record was created or last updated.

Record Stamping - Update

Allowable values for record stamping are:

Attribute	Description / comments
CDTU	Stamped with the date (*SYSFMT8) that the record was last updated.
CYDU	Stamped with the date (CCYYMMDD) that the record was last updated.
DATU	Stamped with the date (*SYSFMT) that the record was last updated.
FUNU	Stamped with the name of the LANSAs function or component that last updated the record.
JNMMU	Stamped with the name of the job that last updated the record.*
JNRU	Stamped with the number of the job that last updated the record.*
PROU	Stamped with the name of the LANSAs Process that last updated the record.
TCDU	Stamped with the DateTime (HHMMSS+*SYSFMT8) that the record was last updated.
TCYU	Stamped with the DateTime (YYYY-MM-DD HH:MM:SS) that the record was last updated.
TDSU	Stamped with the DateTime (HHMMSS+*SYSFMT) that the record was last updated.
TIMU	Stamped with the time (HHMMSS) that the record was last updated.
TYDU	Stamped with the DateTime (HHMMSSYYMMDD) that the record was last updated.
USRU	Stamped with the name of the user that last updated the record.* If *LONG_USER_AUDIT is enabled, then the user name will be the

	audit user (up to 256 characters) if the SET_SESSION_VALUE USER_AUDIT has been set, otherwise the current authenticated user name is used if available, otherwise the current user name is used. If *LONG_USER_AUDIT is not enabled, then the user name will be the audit user (up to 10 characters) if the SET_SESSION_VALUE USER_AUDIT has been set, otherwise the current user name is used.
YMDU	Stamped with the date (YYMMDD) that the record was last updated.

GUI Attributes – Functions Only

Allowable values for GUI Attributes – Functions Only are:

Attribute	Description / comments
CBOX	Present field value as a GUI WIMP Check Box.
DDxx	Drop Down** Represents the field with the corresponding GUI WIMP construct.
PBnn	Push Button **
RBnn	Radio Button **

UD Reporting Attributes – Functions Only

Allowable values for UD Reporting Attributes – Functions Only are:

Attribute	Description / comments
Urxx	Where xx is any alphanumeric combination except for blank characters.

Warnings

- The single asterisk (*) stamping attributes can be used (in the repository only) to indicate that certain fields in file definitions should be automatically

stamped during INSERT and/or UPDATE operations.

- The double asterisk (**) attributes the field with the corresponding GUI WIMP construct. Refer to GUI WIMP Constructs for more information.

Tips & Techniques

- Only one color can be specified for a field.
- Use of colors may affect other attributes.

Platform Considerations

- IBM i: Refer to IBM manual *Data Description Specifications* for more details. Keywords that should be reviewed are COLOR and DSPATR.

Also See

[1.2.17 Input Attributes](#)

[↑ 1.2 Field Definitions](#)

1.2.19 Prompting

Specify the RDML process and function that should be invoked to handle a "prompt request" made against the field being defined or changed. A "prompt request" is made against a field when the user positions the screen cursor into a field, on its label, or on one of its column headings, and then uses the PROMPT function key or equivalent request. Normally the prompt function key is F4, but it may be assigned differently on your system.

Process

Specify the name of the process that has the RDML function to be invoked to handle a "prompt request" made against the field being defined or changed.

Function

Specify the RDML function that should be invoked to handle a "prompt request" made against the field being defined or changed.

Warnings

- LANSAs does not check that the Process or Function exists.

Tips & Techniques

- When specifying the name associated with a prompting process and function, it is recommended that the process name be nominated as *DIRECT. This indicates to the prompt control procedures that the nominated function can be called in "direct" mode, without having to go through the associated process "controller".
- Using *DIRECT has positive performance benefits, but when a prompting function is to be invoked this way it must use the FUNCTION OPTIONS(*DIRECT) command. Refer to [CALL Comments / Warnings](#) and [FUNCTION Examples](#) for more details of direct mode invocation of functions.
- When a reference field has been specified, initially the prompt process/function is inherited from the referenced field, but you can change it if required. Refer to [1.2.7 Reference Field](#).
- If the reference field's prompt process/function is changed, any of the fields referring to it which have the same prompt process/function (before the referenced field is changed) also have their prompt process/function updated.
- For more technical details, refer to [Prompt_Key Processing](#). For examples of prompting processes and functions, please refer to [What Happens When the](#)

[PROMPT Key is Used](#) in the *LANSAs Application Design Guide*.

Also See

[Prompt Key Processing](#)

[↑ 1.2 Field Definitions](#)

1.2.20 Alias Name

Specify the alternate unique name for the field being defined or changed.

Rules

- Must not be the same as any other field's name (including the field that is being created or amended) in the Repository.
- Must not be the same as any other field's alias name in the Repository.
- Maximum of 30 characters.

Tips & Techniques

- COBOL or PL/1 language naming conventions are NOT checked.

Platform Considerations

- IBM i: The alias name facility is provided primarily for installations that use the COBOL or PL/1 programming languages. Refer to the specific IBM supplied program reference manuals for the use of the ALIAS keyword. Name must conform to field naming conventions like the field identifier.

Also See

[1.2.2 Field Identifier](#)

[1.2.1 Field Name](#)

[↑ 1.2 Field Definitions](#)

1.3 Field Visualizations

Field Visualizations can be centrally stored in the LANSAs Repository to provide a consistent visual presentation of a field when used on a Visual LANSAs form or other graphical interface supporting visualizations.

Also see

[Field Visualization Tab](#) in the *User Guide*

[Field Visualization Development](#) in the *Developer Guide*

[Creating Fields](#) in the *Developer Guide*

[↑ 1. Fields](#)

1.4 Field Rules and Triggers

Rules and Triggers can be stored in the LANSAs Repository at both the field level and file level.

It is important to understand how the rules and triggers work at both levels and how the levels work together. Refer to [Rules and Triggers Development Concepts](#) in the *Developer Guide*.

Both field and file level details are found in [Rules and Triggers](#).

Also See

[1.1 Field Types](#)

[Field Rules and Triggers Tab](#) in the *User Guide*

[↑ 1. Fields](#)

1.5 Field Help Text

HELP text is information that is displayed to the user when the LANSAs application requests help (using the HELP key or equivalent request). Help text for fields is stored in the LANSAs Repository. This help text is automatically available as field level context sensitive help text when a field is displayed on a screen. Help text can be entered for each language specified in the partition.

Generally HELP text has the following characteristics:

- It is free format. No restrictions usually exist on the content or format of HELP text.
- It relates directly to the action the user was taking at the time the HELP was requested. Usually the process or function that the user is using is explained in some detail.
- Help text may also include special [Help Text Enhancement & Substitution Values](#).

LANSAs automatically controls the handling of the HELP processing in applications. LANSAs will automatically determine the type of HELP that is required (field, component, process or function) and automatically display the associated HELP text (if any exists).

LANSAs can dynamically, and in the correct language, create the HELP text associated with a field from the repository and the rules that it contains. You can turn off this automatic field level help text feature: globally, by field, or precede it with your own HELP text.

Also See

[Substitution/Control Values](#)

[Substitution/Control Values - Visual LANSAs Only](#)

[Help Text Attributes](#)

[Process Help Text](#)

[Function Help Text](#)

[Repository Help Tab](#) in the *User Guide*

[Repository Help Text Development](#) in the *Developer Guide*

[↑ 1. Fields](#)

2. Rules and Triggers

Rules and triggers are applied at both the field level and file level. The following topics apply to both fields and files in the LANSAs Repository.

[2.1 Rule Definitions](#)

[2.2 List of Values Checks](#)

[2.3 Range of Values Checks](#)

[2.4 Date Format/Range Check](#)

[2.5 Code File/Table Lookup Checks](#)

[2.6 Simple Logic Check](#)

[2.7 Complex Logic Check](#)

[2.8 Triggers](#)

Also See

[Fields](#)

[Files](#)

[File Rules and Triggers Development](#) in the *Developer Guide*

[Field Rules and Triggers tabs](#) in the *User Guide*

[File Rules and Triggers tabs](#) in the *User Guide*.

2.1 Rule Definitions

Rules are defined at both the field level and file level. The following topics apply to both fields and files in the LANSAs Repository.

[2.1.1 Rule Sequence](#)

[2.1.2 Rule Description](#)

[2.1.3 Validation Usage](#)

[2.1.4 Define Rules \(by type\)](#)

[2.1.5 Validation Actions](#)

[2.1.6 Error Message](#)

Also See

[File Rules and Trigger Development](#) in the *Developer Guide*

[Field Rules and Triggers tabs](#) in the *User Guide*

[File Rules and Triggers tabs](#) in the *User Guide*

[2.1.4 Define Rules \(by type\)](#)

[↑ 2. Rules and Triggers](#)

2.1.1 Rule Sequence

Mandatory.

Specify the sequence number for the order to perform the rules.

The sequence number is specific to the rules at the level at which the rule is being added: the field level or file level. Field levels are applied before the file level rules.

Rules

- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a rule is added. Sequence number is updated when the order of the rules is updated in the list of rules.

Tips & Techniques

- The rule sequence number, or order that the rules are processed, is extremely important as you review the [2.1.5 Validation Actions](#) performed for each rule.

Also See

[Understanding Rule Sequence](#) in the *Developer Guide*

[Rule Hierarchy](#) in the *Developer Guide*

[↑ 2.1 Rule Definitions](#)

2.1.2 Rule Description

Mandatory.

Specify a brief description of the rule to aid other developers in understanding its purpose.

Rules

- Description is a maximum of 30 characters.

[↑ 2.1 Rule Definitions](#)

2.1.3 Validation Usage

When inserting

Mandatory. Default= Always apply rule (ADD)

Specify database operation when the rule is to be performed.

Rules

Allowable values are:

Always apply rule (ADD)	Rule is always applied when information is added (inserted) to the database.
Apply when field is used (ADDUSE)	Rule is only applied when the field is actually specified in the INSERT command being executed.
Never apply rule	Do not apply rule when inserting to the file.

When updating

Mandatory. Default= Always apply rule (CHG)

Specify database operation when the rule is to be performed.

Rules

Allowable values are:

Always apply rule (CHG)	Rule is always applied when information is changed (updated) in the database.
Apply when field is used (CHGUSE)	Rule is only applied when the field is actually specified in the UPDATE command being executed.
Never apply rule	Do not apply rule when updating the file.

When deleting

Mandatory. Default= Never apply rule

Specify database operation when the rule is to be performed.

Rules

Allowable values are:

Always apply rule (DLT)	Rule is always applied when information is deleted (removed) from the database.
Never apply rule	Do not apply rule when deleting from the file.

Tips & Techniques

- Most commonly used entries are ADD, CHG and CHGUSE.
- Use of DLT by itself is a common and a very powerful rule mechanism.
- If ADDUSE is specified, ensure that the default value of the field is a valid database value.
- Use caution when specifying CHGUSE with a rule that involves multiple fields, because the check will only be done when the field linked to the rule is specified on an UPDATE command, and not done when it is omitted, regardless of whether or not any of the other fields referenced in the rule are specified.
- When creating a rule, ensure that it does not indirectly interfere with a trigger. For more information, refer to [Triggers - Some Do's and Don'ts](#).

Also See

[2.1.4 Define Rules \(by type\)](#)

[↑ 2.1 Rule Definitions](#)

2.1.4 Define Rules (by type)

The fields to be specified are dependent upon the type of validation rule. Refer to:

[2.2 List of Values Checks](#)

[2.3 Range of Values Checks](#)

[2.4 Date Format/Range Check](#)

[2.5 Code File/Table Lookup Checks](#)

[2.6 Simple Logic Check](#)

[2.7 Complex Logic Check](#)

[↑ 2.1 Rule Definitions](#)

2.1.5 Validation Actions

If Field is Within the Checks

Mandatory. Default=Evaluate next rule (NEXT).

Specify action to be performed if the field is found to be in one of the ranges specified.

Rules

Allowable values are:

Evaluate next rule (NEXT)	Field is okay. Proceed to next rule for this field. No error message is displayed.
Set field in error (ERROR)	Field is in error. Issue error message described below.
Value is accepted (ACCEPT)	Field is okay. Bypass all other rules for this field.

If Field is Not Within the Checks

Mandatory. Default=Set field in error (ERROR).

Specify action to be performed if the field is NOT found to be in one of the ranges specified.

Rules

Allowable values are:

Evaluate next rule (NEXT)	Field is okay. Proceed to next rule for this field. No error message is displayed.
Set field in error (ERROR)	Field is in error. Issue error message described below.
Value is accepted (ACCEPT)	Field is okay. Bypass all other rules for this field.

Tips & Techniques

- When using ACCEPT to bypass all other rules for this field, the Order to Process rules are very important. Refer to [2.1.1 Rule Sequence](#).

Also See

[2.1.4 Define Rules \(by type\)](#)

[↑ 2.1 Rule Definitions](#)

2.1.6 Error Message

Specify either message number and message file or message text. If neither an error message number and file nor error message text is specified LANSAs will insert a default error message number, file and library as the error message. These default messages are "general purpose" and do not provide much detail about the specific cause of the error.

Message File

Specify the message file name to be used to locate the identified message number.

Message file is mandatory if using a Message Number.

Platform Considerations

- IBM i: Error message files and error message numbers are a native part of the IBM i operating system. Refer to the IBM supplied Control Language Reference Manual for more details. CL commands involving message files include CRTMSGF and ADDMSGD.
- IBM i: You can directly edit the message details from this screen panel. Enter as much of the message details as is known and use the function key labeled "Work Msgd" (Work Message Description). This will cause an IBM i WRKMSGD command to be executed, using as much of the supplied message details as is possible. This IBM i facility will allow you to create or edit the message details. Upon completion of the WRKMSGD command, this screen panel will be redisplayed, unchanged, to allow you to proceed.
- Do not store user defined messages, or modify "shipped" messages, in the LANSAs message file DC@M01 via this or any other message file editing facility. It is regularly replaced by new versions or EPCs (Expedited Program Changes).

Error Message Number

Specify the message number from the identified message file that you wish displayed.

Message number is mandatory if using a Message File.

Error Message Text

Specify the text of the error message to be displayed directly.

Message length can be a maximum of 80 characters.

Tips & Techniques

- If Message Text facility is used, then the message will have no second level text associated with it.
- Special substitution characters can be used in the message.

[↑ 2.1 Rule Definitions](#)

2.2 List of Values Checks

A list of values rule allows a field to be checked against a list of values. For instance, an Australian state should be in the list QLD, NSW, VIC, etc.

You can modify the sequencing of the rule by changing the [2.1.1 Rule Sequence](#) field. A [2.1.2 Rule Description](#) must be associated with the rule. All the currently existing values for the rule are shown in a list.

Refer to:

[2.2.1 Value](#)

[2.2.2 List Examples](#)

Also See

[2.1.4 Define Rules \(by type\)](#)

[↑ 2.1 Rule Definitions](#)

2.2.1 Value

Specify the value to add to the values list or change in the value list.

At least one entry in the list is required, and a maximum of 50 entries can be specified. Values are checked for type and length compatibility.

Rules

Allowable values include:

- An alphanumeric literal (in quotes) such as 'NSW', 'BALMAIN'
- A numeric literal such as 1, 14.23, -1.141217.
- Another field name such as CUSTNO, INVNUM, etc.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.
- A process parameter such as *UP01, *UP02, etc.

Tips & Techniques

- You may wish to use [2.3 Range of Values Checks](#) depending upon your list contents. For large numeric lists, a range check may be more appropriate.

[↑ 2.1 Rule Definitions](#)

2.2.2 List Examples

These examples are provided to illustrate the use of the list of values rule facility:

Example 1

Field being checked:

Name	Type	Len	Dec
STATE	A	3	

List of values:

List Of Values	Comments
'NSW' 'QLD' 'VIC'	Check for valid Australian state mnemonic.

Example 2

Field being checked:

Name	Type	Len	Dec
NAME	A	7	

List of values:

List Of Values	Comments
*BLANKS	A blank name is an error. Reverse the default error logic to get ERROR if in list, NEXT if not in list.

Example 3

Field being checked:

Name	Type	Len	Dec
COMPNO	P	1	0

List of values:

List Of Values	Comments
1	Company number must be 1, 2 or 3.
2	
3	

Example 4

Field being checked:

Name	Type	Len	Dec
YEAR	A	2	

List of values:

List Of Values	Comments
*LASTYEAR	Year must equal one of the site defined system variables *LASTYEAR, *THISYEAR or *NEXTYEAR
*THISYEAR	
*NEXTYEAR	

[↑ 2.2 List of Values Checks](#)

2.3 Range of Values Checks

Specify a field to be checked against various ranges of values.

A Range of Values rule allows a field to be checked against various ranges of values. For instance, an Australian postcode can be in one of the ranges 2000 - 2999, 3000 - 3999, etc.

You can modify the sequencing of the rule by changing the [2.1.1 Rule Sequence](#) field. A [2.1.2 Rule Description](#) must be associated with the rule. All the currently existing values for the rule are shown in a list.

Refer to:

[2.3.1 Range: From value / To value](#)

[2.3.2 Range of Values Examples](#)

Also See

[2.1.4 Define Rules \(by type\)](#)

[↑ 2.1 Rule Definitions](#)

2.3.1 Range: From value / To value

Mandatory.

Specify a "from" and "to" value for each range that the field will be checked against.

At least one range is required and up to a maximum of 20 ranges can be specified. The range values are checked for type and length compatibility.

Rules

Allowable values are:

- An alphanumeric literal (in quotes) such as 'NSW', 'BALMAIN'.
- A numeric literal such as 1, 14.23, -1.141217.
- Another field name such as CUSTNO, INVNUM, etc. Notice that a field name is not entered as #CUSTNO.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.
- A process parameter such as *UP01, *UP02, etc.
- System does NOT check that the "from" value is less than "to" value.

Tips & Techniques

- You may wish to use [2.2 List of Values Checks](#) when a small range of values are being checked.

Also See

[2.3.2 Range of Values Examples](#)

[↑ 2.3 Range of Values Checks](#)

2.3.2 Range of Values Examples

These examples illustrate the use of the range of values rule facility:

Example 1

Field being checked:

Name	Type	Len	Dec
POSTCD	A	4	

Range of values:

	From Value	To Value	Comments
Or	'2000'	'2900'	Check post code is in NSW, VIC or QLD. Post code is numeric.
Or	'3000'	'3900'	
Or	'4000'	'4900'	

Example 2

Field being checked:

Name	Type	Len	Dec
POSTCD	S	4	0

Range of values:

	From Value	To Value	Comments
Or	2000	2900	Check post code is in NSW, VIC or QLD. Post code is numeric.
Or	3000	3900	

Or	4000	4900	
----	------	------	--

Example 3

Field being checked:

Name	Type	Len	Dec
AMOUNT	P	5	2

Range of values:

From Value	To Value	Comments
0.01	999.99	Check that AMOUNT is positive.

Example 4

Field being checked:

Name	Type	Len	Dec
AMOUNT	P	5	2

Range of values:

From Value	To Value	Comments
-999.99	-0.01	Check that AMOUNT is negative.

Example 5

Field being checked:

Name	Type	Len	Dec
PERIOD	A	2	

Range of values:

From Value	To Value	Comments
*LASTPER	*NEXTPER	Check PERIOD using site defined system variables *LASTPER and *NEXTPER.

[↑ 2.3 Range of Values Checks](#)

2.4 Date Format/Range Check

A date format/range rule allows a field to be validated as a date in a certain format (DDMMYY, MMDDYY, etc.) and in a certain range. For instance, a "date order due" may have to be in format DDMMYY and no more than 90 days into the future.

You can modify the sequencing of the rule by changing the [2.1.1 Rule Sequence](#) field. A [2.1.2 Rule Description](#) must be associated with the rule.

Refer to:

[2.4.1 Date Format](#)

[2.4.2 Number of Days Allowed into the Past](#)

[2.4.3 Number of Days Allowed into the Future](#)

[2.4.4 Date Format/Range Check Examples](#)

Also See

[2.1.4 Define Rules \(by type\)](#)

[↑ 2.1 Rule Definitions](#)

2.4.1 Date Format

Mandatory. Default=SYSFMT.

Rules

Allowable values are:

SYSFMT	operating system date format (from QDATFMT)
DDMMYY	day month year format
MMDDYY	month day year format
YYMMDD	year month day format
DDMMYYYY	day month century year format
MMDDYYYY	month day century year format
YYYYMMDD	century year month day format
YYYYDDMM	century year day month format
YYMM	year month format
MMYY	month year format
YYYYMM	century year month format
MMYYYY	month century year format
SYSFMT8	operating system date format including century

Date Format Examples

For example, the date 28th October 1986 would have to be entered as follows to satisfy each format type:

SYSFMT	281086 (Usual format for Australia and Europe)
SYSFMT	102886 (Usual format for USA)
DDMMYY	281086
MMDDYY	102886

YYMMDD	861028
DDMMYYYY	28101986
MMDDYYYY	10281986
YYYYMMDD	19861028
YYYYDDMM	19862810
YYMM	8610
MMYY	1086
YYYYMM	198610
MMYYYY	101986
SYSFMT8	28101986 (Usual format for Australia and Europe)
SYSFMT8	10281986 (Usual format for USA)

Tips & Techniques

- In a client/server application, the client's date format will be automatically passed to the server. If the client and server date formats are different (e.g. MDY vs DMY), the server will automatically return data in the client's format.
- The client's format can be changed from the default by specifying the X_RUN parameter DATF=. For more information, please refer to [Standard X_RUN Parameters](#).
- If client and server date formats are different (such as between USA and UK clients), date format validation rules specifying exact formats will fail. For example, DDMMYY may be returned as MMDDYY. Where clients need to use different date formats, date format SYSFMT is recommended.

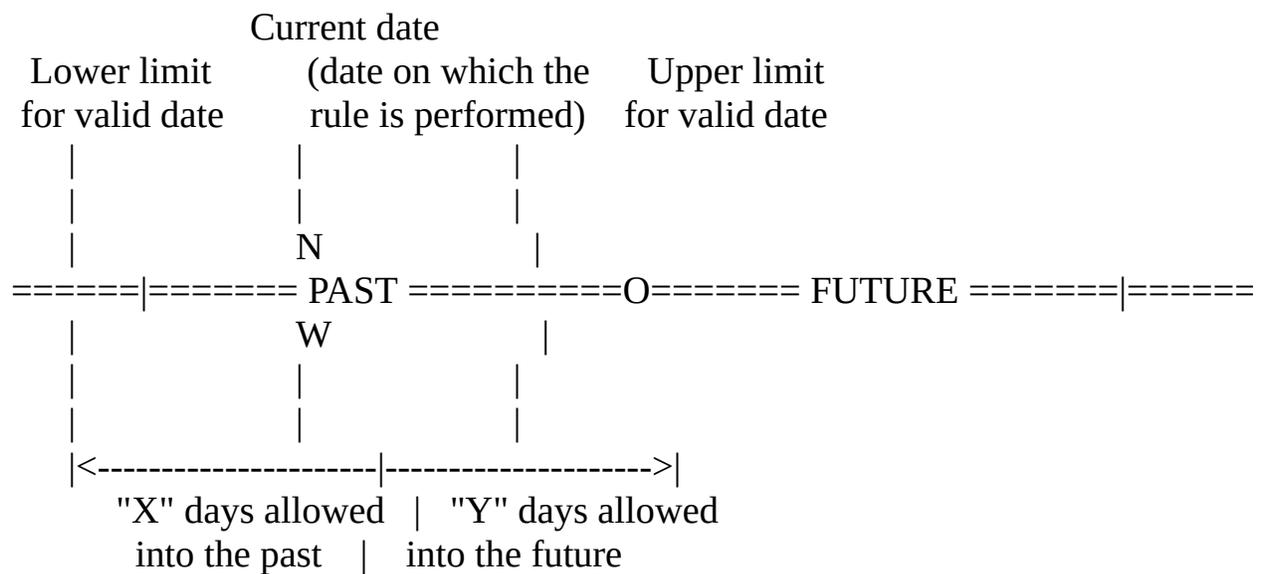
↑ [2.4 Date Format/Range Check](#)

2.4.2 Number of Days Allowed into the Past

Mandatory. Default=9999999

Specify the lower limit of the date range rule.

The use of the "days into the past" and "days into the future" range limit values can be illustrated with a time line:



Tips & Techniques

- If you are simply performing a date format check, leave this value as 9999999 so that there are no range limits applied.

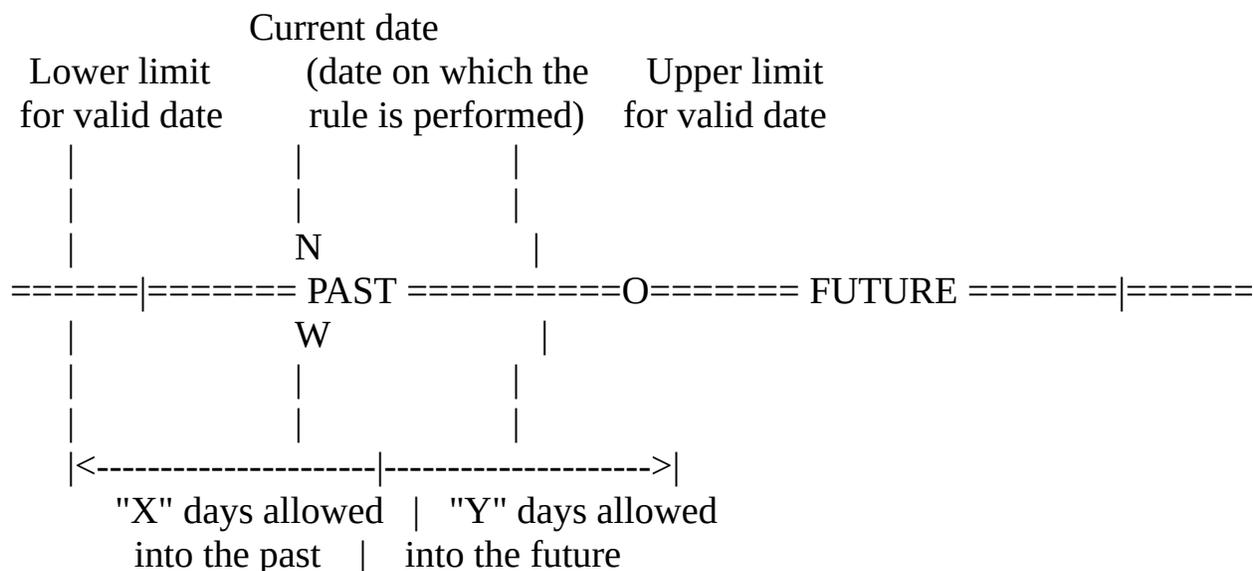
[↑ 2.4 Date Format/Range Check](#)

2.4.3 Number of Days Allowed into the Future

Mandatory. Default=9999999

Specify the higher limit of the date range rule.

The use of the "days into the past" and "days into the future" range limit values can be illustrated with a time line:



Tips & Techniques

- If you are simply performing a date format check, leave this value as 9999999 so that there are no range limits applied.

[↑ 2.4 Date Format/Range Check](#)

2.4.4 Date Format/Range Check Examples

These examples are provided to illustrate the use of the date format/range rule:

Description of Check Required	Date Format	Days Into Past	Days Into Future
Check field DATDUE is in format DDMMYY	DDMMYY	9999999	9999999
Check field DATDUE is in format YYMMDD and is not prior to the current date.	YYMMDD	0	9999999
Check field DATE is in format DDMMYYYY and is within the next 90 days.	DDMMYYYY	0	90
Check field DATE is in format DDMMYY and is in the last 180 days.	DDMMYY	180	0
Check field DATE is in format YYYYMMDD and is in the last week.	YYYYMMDD	7	0
Check field DATE is in format YYYYDDMM and is within the next week.	YYYYDDMM	0	7
Check field DATE is in format YYMM.	YYMM	9999999	9999999
Check field DATE is in format MMDDYYYY and is within the next 30 days.	MMDDYYYY	0	30

[↑ 2.4 Date Format/Range Check](#)

2.5 Code File/Table Lookup Checks

A code/table file lookup rule allows a field to be "looked up" in a code file or table. For instance, a product number may be looked up in the product master file to check that it is a valid number.

You can modify the sequencing of the rule by changing the [2.1.1 Rule Sequence](#) field. A [2.1.2 Rule Description](#) must be associated with the rule.

A code/table file lookup is typically defined for file level validation checks only.

Refer to:

[2.5.1 Lookup File Name](#)

[2.5.2 Key Field or Literal](#)

[2.5.3 Code File/Table Lookup Check Examples](#)

Also See

[2.1.4 Define Rules \(by type\)](#)

[↑ 2.1 Rule Definitions](#)

2.5.1 Lookup File Name

Mandatory.

Specify the physical or logical file / table that is to be used for checking.

Rules

- File name must exist in the Repository.

Tips & Techniques

- You can prompt for all the file definitions currently existing in the repository.

[↑ 2.5 Code File/Table Lookup Checks](#)

2.5.2 Key Field or Literal

Specify the key that is to be compared with the keyed index of the file looking for a "match" (i.e.: a record with an identical key in the file).

At least 1 key field is required. The entire key list supplied is checked for compatibility with the actual key(s) of the file. The key list specified can be a full or partial key to the file. A warning is issued if a partial key list is specified.

Rules

Allowable values are:

- For field level validations, the name of the current field.
- For file level validations, a field from the current file.
- An alphanumeric literal (in quotes) such as 'NSW', 'BALMAIN'
- A numeric literal such as 1, 14.23, -1.141217.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.
- A process parameter such as *UP01, *UP02, etc.

Tips & Techniques

- A code/table file lookup is typically defined for file level validation checks only.

[↑ 2.5 Code File/Table Lookup Checks](#)

2.5.3 Code File/Table Lookup Check Examples

These examples are provided to illustrate the use of the code/table file lookup rule facility:

Example 1

Field being checked:

Name	Type	Len	Dec
PRODNO	A	10	

File details:

Name	Actual Keys	Keys Supplied	Comments
PRODMST	Product number	PRODNO	Check product is in product master

Example 2

Field being checked:

Name	Type	Len	Dec
TAXCDE	A	3	

File details:

Name	Actual Keys	Keys Supplied	Comments
TAXTABL	Tax code type	INCOME	Check TAXCDE is a valid income tax code using alpha literal 'INCOME' in the key list.
	Tax	TAXCDE	

	code		
--	------	--	--

Example 3

Field being checked:

Name	Type	Len	Dec
PARTNO	P	7	0

File details:

Name	Actual Keys	Keys Supplied	Comments
INVENT	Part number Warehouse number	PARTNO	Check PARTNO exists. This is a "partial" key validation check.

Example 4

Field being checked:

Name	Type	Len	Dec
INVNUM	A	8	

File details:

Name	Actual Keys	Keys Supplied	Comments
INVNUM	Invoice number	INVNUM	Check INVNUM does not already exist. Reverse default error logic to get ERROR if key match is found, NEXT if key match is NOT found.

↑ 2.5 Code File/Table Lookup Checks

2.6 Simple Logic Check

A simple logic rule allows simple conditions to be evaluated to check a field. For instance, "item weight must be less than (item volume * 10.643)" may be a check used in an order entry system.

You can modify the sequencing of the rule by changing the [2.1.1 Rule Sequence](#) field. A [2.1.2 Rule Description](#) must be associated with the rule.

Refer to:

[2.6.1 Condition to Check](#)

[2.6.2 Simple Logic Check Examples](#)

Also See

[2.1.4 Define Rules \(by type\)](#)

[↑ 2.1 Rule Definitions](#)

2.6.1 Condition to Check

Mandatory.

Specify an expression that can be evaluated and produces a "true" or "false" result.

The expression must be syntactically correct. Expression evaluation is left to right within brackets, so use brackets whenever in doubt as to the order in which the expression will be evaluated. Expression components are checked for type and length compatibility. Error messages are issued that indicate any problems found when attempting to evaluate the expression.

Rules

Valid expression operators are:

(Open bracket)	Close bracket
+	Add	-	Subtract
/	Divide	*	Multiply
=	Compare equal	^=	Compare not equal
*EQ	Compare equal	*NE	Compare not equal
		*NE=	Compare not equal
<	Compare less than	>	Compare greater than
*LT	Compare less than	*GT	Compare greater than
<=	Compare less than or equal to	>=	Compare greater than or equal to
*LE	Compare less than or equal to	*GE	Compare greater than or equal to
AND	And	OR	Or

Components of the expression can be:

- An alphanumeric literal such as 'NSW', NSW, 'Balmain' or BALMAIN
- A numeric literal such as 1, 14.23, -1.141217.
- Another field name such as #CUSTNO, #INVNUM, etc.

- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.
- A process parameter such as *UP01, *UP02, etc.
- Alphanumeric literals do NOT have to be in quotes when used in an expression. Quotes are only required when the alphanumeric literal contains lowercase characters. If no quotes are used the alpha literal is converted to uppercase. Thus BALMAIN = balmain = Balmain = balMAIN. However, Balmain does not equal 'Balmain'.
- Field names must be preceded by a # (hash) symbol when used in expressions. This allows LANSAP to differentiate between fields and alphanumeric literals. For instance the expression CNTRY = AUST does not indicate which of the components is the field and which is the alphanumeric literal. The correct format is #CNTRY = AUST or #CNTRY = 'AUST'.

[↑ 2.6 Simple Logic Check](#)

2.6.2 Simple Logic Check Examples

These examples illustrate the use of the simple logic rule facility:

Check Required	Actual Expression Used
Field VALUE must be greater than zero	#VALUE > 0 or #VALUE *GT 0
Field STATE must be NSW, VIC or QLD.	(#STATE = NSW) OR (#STATE = VIC) (#STATE = QLD)
Field WEIGHT must be zero if field MEASUR is not zero.	(#MEASUR ^= 0) AND (#WEIGHT = 0) or (#MEASUR *NE 0) *AND (#WEIGHT *EQ 0)
Field WEIGHT must be less than field MEASUR multiplied by 10.462	#WEIGHT < (#MEASUR * 10.462) or #WEIGHT *LT (#MEASUR * 10.462)

[↑ 2.6 Simple Logic Check](#)

2.7 Complex Logic Check

A complex logic rule allows complex validation checking to be performed by your own LANSAs or 3GL application programs. For instance, the validation of a "due date" may be done via a function or program that can account for public holidays, weekends, etc.

You can modify the sequencing of the rule by changing the [2.1.1 Rule Sequence](#) field. A [2.1.2 Rule Description](#) must be associated with the rule.

Refer to:

[2.7.1 Program to Call](#)

[2.7.2 Program to Call: Function](#)

[2.7.3 Program to Call: 3GL Program](#)

[2.7.4 3GL Parameters](#)

[2.7.5 Complex Logic Check Examples](#)

Also See

[2.1.4 Define Rules \(by type\)](#)

*Technical notes for *ALP_FIELD_VALIDATE and *NUM_FIELD_VALIDATE in [Function Parameters](#).*

[↑ 2.1 Rule Definitions](#)

2.7.1 Program to Call

Mandatory. Default=3GL Program.

Specify whether a LANSAP function or 3GL program is to be called.

Rules

Allowable values are:

- Function
- 3GL Program

Tips & Techniques

- No check is done for the existence of the function/program.

Also See

[2.7.2 Program to Call: Function](#)

[2.7.3 Program to Call: 3GL Program](#)

[↑ 2.7 Complex Logic Check](#)

2.7.2 Program to Call: Function

Specify that a LANSAs function is to perform the validation check.

Rules

- Mandatory if Function is specified in [2.7.1 Program to Call](#).

Tips & Techniques

- LANSAs does not check whether the function specified exists or not.
- See the [FUNCTION](#) command for design constraints on validation functions.

Platform Considerations

- IBM i: The function specified should be found in your library list at the time the rule is to be performed.

Also See

[2.7.1 Program to Call](#)

[↑ 2.7 Complex Logic Check](#)

2.7.3 Program to Call: 3GL Program

Specify that a 3GL program is to perform the validation check.

Rules

- Mandatory if 3GL Program is specified in [2.7.1 Program to Call](#).

Tips & Techniques

- LANSAs do not check whether the program specified exists or not.

Platform Considerations

- IBM i: The program specified should be found in your library list at the time the rule is to be performed.

Also See

[2.7.1 Program to Call](#)

[↑ 2.7 Complex Logic Check](#)

2.7.4 3GL Parameters

Optional.

Specify additional parameters, if any, to be passed to the program.

Additional parameters may only be used when a 3GL program is called, and cannot be used when a validation function is called.

Rules

Additional parameters may be:

- An alphanumeric literal (in quotes) such as 'NSW', 'BALMAIN'.
- A numeric literal such as 1, 14.23, -1.141217.
- Another field name such as CUSTNO, INVNUM, etc.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.
- A process parameter such as *UP01, *UP02, etc.
- For alphanumeric fields (alpha literals, alpha fields, alpha system variables or alpha process parameters), the parameter is passed as alpha (256) with the parameter value left aligned into the 256 byte parameter.
- For numeric fields (numeric literals, numeric fields, numeric system variables or numeric process parameters), the parameter is passed as packed 15 with the same number of decimal positions as the parameter value. For numeric literals, this means the same number of decimal positions as specified in the literal (e.g.: 1.12 will be passed as packed 15,2. 7.12345 will be passed as packed 15,5. 143 will be passed as packed 15,0. etc.). For all other types of numeric parameters, this means the same number of decimal positions as their respective definitions.
- The type and length of the parameter(s) passed depends upon the type and length of the parameter value supplied.
- As with the standard parameters, the actual value is passed in a work area so it is not possible to change the value of a field by changing the parameter value in the validation program.
- LANSAs does not check that the 3GL program being called has the correct parameter list. This is your responsibility.

All 3GL programs called as part of a complex logic rule must have a least 3 standard parameters. These are:

Name Description

Return code Alphanumeric length 1. Returned by the program as '1' (good return) or '0' (bad return). Used by the program to indicate to LANSAs the success or failure of the complex logic rule.

Name of field Alphanumeric length 10. Passed to the program. Contains the name (as opposed to the value) of the field that is passed in the third parameter.

Value of field Length and type depend upon the repository definition of the field. Alphanumeric fields are passed with same type and length as their repository definition. All numeric fields (type P or S) are passed as packed (type P) and the same length and number of decimal positions as their repository definition. Note that the value of the field is passed in a work area, thus it is not possible to change the value of the field by changing the value of the parameter in the validation program.

Also See

[2.7.3 Program to Call: 3GL Program](#)

[↑ 2.7 Complex Logic Check](#)

2.7.5 Complex Logic Check Examples

These examples are provided to illustrate the use of the complex logic rule facility:

Check Required	Program Specified	Additional Parameters
Pass field DATDUE to program DATECHECK for validation.	DATECHECK	none
Pass field DATDUE to program INVCHECK for validation. Also pass fields INVNUM and COMPNO.	INVCHECK	INVNUM COMPNO
Pass field TAXCDE to program TAXCHECK for validation. Also supply tax scale 7 as a parameter with 3 decimal positions.	TAXCHECK	7.000 or +7.000 or 07.000

See also example *Using the CALLCHECK Command for Inline Validation* in the [CALLCHECK Examples](#).

[↑ 2.7 Complex Logic Check](#)

2.8 Triggers

A trigger function is a type of LANSAs function which will be invoked **automatically** when a specific type of operation occurs and when a specific set of conditions are met. Triggers are applied at both the field level and file level. The following topics apply to both fields and files in the LANSAs Repository.

[2.8.1 Trigger Definition](#)

[2.8.2 Trigger Condition](#)

[2.8.3 Trigger Functions](#)

Also See

[Trigger Concepts](#) in the *Developer Guide*

[File Rules & Triggers Development](#) in the *Developers Guide*

[Field Rules and Triggers tab](#) in the *User Guide*

[File Rules and Triggers tab](#) in the *User Guide*

[↑ 2. Rules and Triggers](#)

2.8.1 Trigger Definition

Triggers are defined at both the field level and file level. The following topics apply to both fields and files in the LANSAs Repository.

[Trigger Description](#)

[Trigger Function Name](#)

[Trigger Points](#)

[Trigger Definition Example](#)

Also See

[File Rules and Triggers Development](#) in the *Developer Guide*

[Field Rules and Triggers tabs](#) in the *User Guide*

[File Rules and Triggers tabs](#) in the *User Guide*

[↑ 2.8 Triggers](#)

Trigger Description

Mandatory. Default=New Trigger

Specify a brief description of the trigger to aid other developers in understanding its purpose.

Rules

- Description is a maximum of 30 characters.

[↑ 2.8.1 Trigger Definition](#)

Trigger Function Name

Mandatory.

Specify the name of the trigger function which will be called when the trigger conditions are satisfied.

Rules

- The named function must be defined with the TRIGGER parameter on the FUNCTION command.
- LANSA does not check whether the function specified exists or not.

Tips & Techniques

- Review the [FUNCTION](#) command for additional information about trigger functions.
- [File Rules & Triggers Development](#) in the *Administrator Guide*.

[↑ 2.8.1 Trigger Definition](#)

Trigger Points

Mandatory.

Specify when the function is to be triggered.

At least one trigger point must be specified.

Rules

Allowable values are:

Option	Triggered
Before Open	Before the file is opened.
Before Close	Before the file the field is in is closed.
Before Read	Before the file the field is in is read.
Before Insert	Before the file the field is in is inserted.
Before Update	Before the file the field is in is updated.
Before Delete	Before the file the field is in is deleted.
After Open	After the file the field is in has been opened.
After Close	After the file the field is in has been closed.
After Read	After the file the field is in has been opened (accessed).
After Insert	After the file the field is in has been opened (accessed).
After Update	After the file the field is in has been opened (accessed).
After Delete	After the file the field is in has been opened (accessed).

[↑ 2.8.1 Trigger Definition](#)

Trigger Definition Example

Trigger field:

Name	Type	Len	Dec
SALARY	P	12	2

Trigger function:

Name	Function	Trigger Before	Points After	Comments
TRGFN1	Open			Call trigger function TRGFN1 following an insert or update of field SALARY, or before a delete of field SALARY. Note that no conditions have been applied to this trigger.
	Close			
	Read			
	Insert		Y	
	Update		Y	
	Delete	Y		

Also See

[File Rules & Triggers Development](#) in the *Developers Guide* for details and examples.

[↑ 2.8.1 Trigger Definition](#)

2.8.2 Trigger Condition

The use of a trigger condition is optional. If no test conditions are specified, the trigger will be unconditionally invoked. No conditions are allowed if "Open" or "Close" [Trigger Points](#) are used.

If the trigger is defined at field level, the conditions can only involve the field to which the trigger is being linked. If the trigger is defined at the file level, the conditions can involve any other real or virtual fields in the file to which the trigger is being linked.

Refer to:

[And/Or Logic](#)

[Field Name](#)

[Operator](#)

[Compare to value](#)

[Trigger Sequence](#)

Also See

[Trigger Concepts](#) in the *Developer Guide*

[↑ 2.8 Triggers](#)

And/Or Logic

Mandatory. Default=AND

Specify whether multiple conditions are joined with a logical "AND" or "OR".

If only one condition is entered, this value is ignored.

Rules

Allowable values are:

- AND
- OR

[↑ 2.8 Triggers](#)

Field Name

Mandatory.

Specify the field name to be used on the condition expression.

Rules

- Field must exist in the current file.

[↑ 2.8 Triggers](#)

Operator

Mandatory. Default=Equal to

Specify the operator for the condition that will be used with the [Field Name](#) and [Compare to value](#).

Rules

Operations allowed are:

GT Greater than

GE Greater than or equal to

LT Less than

LE Less than or equal to

EQ Equal to

NE Not equal to

REF Refers to

Tips & Techniques

- All operation codes can be suffixed by a "P" indicating that the previous value of the nominated field should be used. Such conditions are only valid for "Update" operations. Also the [Compare to value](#) of the condition must not be a literal if this type of operation code is used.
- The "REF" operation code is only valid for "Insert", "Update" and "Read" operations.

[↑ 2.8 Triggers](#)

Compare to value

Mandatory.

Specify the field/literal that will make up the value which is being compared with the [Field Name](#).

The values are checked for type and length compatibility with the [Field Name](#).

If the operation codes are suffixed by a "P", the condition must not be a literal.

Rules

Allowable values are:

- An alphanumeric literal such as 'NSW', NSW, 'Balmain' or BALMAIN
- A numeric literal such as 1, 14.23, -1.141217.
- Another field name such as #CUSTNO, #INVNUM, etc.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.

[↑ 2.8 Triggers](#)

Trigger Sequence

Mandatory. Default=Next sequential number.

Specify the sequence in which the statements are to be executed.

Rules

- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a condition is added.

Tips & Techniques

- Sequence of the conditions is important when setting the trigger [And/Or Logic](#).

[↑ 2.8 Triggers](#)

2.8.3 Trigger Functions

[What is a Trigger Function?](#)

[Create a Trigger Function](#)

[Activate a Trigger Function](#)

[Exactly When Are Triggers Invoked?](#)

[The TRIG_OPER and TRIG_RETC Variables and TRIG_LIST Working List](#)

[What Codes Are Passed in TRIG_OPER to the Trigger?](#)

[How Many Entries Are Passed in the TRIG_LIST?](#)

[Triggers - A Classic Example](#)

[Examples of Trigger use](#)

[Triggers - Restrictions and Limitations](#)

[Triggers - Some Do's and Don'ts](#)

[↑ 2.8 Triggers](#)

What is a Trigger Function?

A trigger function is a type of LANSAs function which will be invoked **automatically** when a specific type of I/O operation occurs to a file and when a specific set of conditions are met.

For example, when an application developer defines the "Cancel of an Order" via the RDML command DELETE FROM_FILE(ORDHDR) WITH_KEY(#ORDNUM), they have initiated an "event", which may automatically cause other functions to be "triggered".

When the order is canceled it may "trigger" the following:

Activity A Flag Order Historical Details

Activity B Print Outstanding Credit Invoices

Activity C Send a Message to the Sales Department

Additionally, the list of activities that happen when an order is canceled can be added to or changed at **any time** without having to change or even recompile the original DELETE FROM_FILE(ORDHDR) function.

A trigger function allows a business activity to be associated directly with a database file (i.e. the "object"). When a specified event happens to information in the file, then the trigger(s) will be automatically invoked.

For example, if the business rules stated that when an order is canceled you must also perform activities A, B and C, then a "traditional design" would include A, B and C as direct logic (or calls) into the interactive function called "Cancel an Order".

In fact there may be several sources from which an Order may be canceled:

Source The typical interactive "Cancel an Order" transaction.

1

Source Monthly Batch Automatic Canceling of Unfilled Orders.

2

Source Requests arriving via LANSAs Open transactions from sales people using dial up PC systems.

3

and the most "dangerous" source of all:

Source X The transaction that will be defined by someone else in 2 years time.

You have to remember to include the A, B and C activities (or at least the initiation of it) into sources 1 through 3 now.

In fact it is often the last case, "Source X", that will cause the most problem when the new designer fails to realize that the A, B and C logic exists, or even to understand fully how and when it is used.

Triggers solve all these problems because they link the activities A, B and C to the "object" Order, and thus are always invoked, no matter what the "source" of the event is.

And best of all, you can add new activities D, E and F to Order, at any time, without having to change any of the event "sources" in any way.

Sensible use of triggers may transform the way that an application is designed. The user interface can be fully designed, and then the complexities and rules can be introduced later by using data dictionary validation rules and database trigger functions.

The resulting design is much more in the "object oriented" style.

Triggers separate "business function" from "user interface" in a much clearer and easier way.

[↑ 2.8.3 Trigger Functions](#)

Create a Trigger Function

To define a function as being a trigger function, use the TRIGGER parameter on the FUNCTION command.

If the function is to act as a data dictionary level trigger, enter *FIELD into the first part of the parameter, and the associated data dictionary field name into the second part of the parameter.

If the function is to act as a database level trigger, enter *FILE into the first part of the parameter, and the associated database file name into the second part of the parameter. The file specified must be a physical file.

As an aid to defining new trigger functions LANSAs is shipped with the following Application Templates that can be used to form the base of a trigger function:

BBFLDTRIG Field Level Trigger Function

BBFILTRIG File Level Trigger Function

When a function is defined as a trigger function you **must** follow these guidelines:

- The parameter RCV_LIST(#TRIG_LIST) must be used.
- The parameter RCV_DS must not be used.
- Option *DIRECT must also be used.
- Options xxx_SYSTEM_VARIABLE or xxx_FIELD_VALIDATE must not be used.
- The list #TRIG_LIST must be defined by a DEF_LIST command as DEF_LIST NAME(#TRIG_LIST) TYPE(*WORKING) ENTRIES(2) and must not include any fields in the FIELDS parameter. The required fields will be automatically added.
- No DISPLAY, REQUEST or POP_UP commands may be used. This is a deliberately imposed design/usage constraint that may be removed in later versions.
- No CALL can exist to another process/function. This is a deliberately imposed design/usage constraint that may be removed in later versions.
- Trigger functions cannot be defined within an action bar process. This is not to say that they cannot be referenced from within an action bar, it just means that a trigger function cannot be defined as part of a process that is of action

bar type.

- The associated process must not have any parameters.
- The exchange list may not be used. This is a deliberately imposed design/usage constraint imposed to enforce insulated and modular design and use of trigger functions.

When a function is defined as a trigger function you should follow these guidelines in most situations:

- Understand how triggers are defined and how they should be used by reading the [Field Rules/triggers](#) sections and Trigger Functions section.
- Use options *NOMESSAGES and *MLOPTIMIZE.
- Options *HEAVYUSAGE and *DBOPTIMIZE may also be considered.
- Do not directly or indirectly access the database file that the trigger is, or will be, linked to.
- Where triggers are heavily and constantly invoked avoid resource intensive operations. Such operations will slow down access to the associated file. Whenever reasonable make the trigger "submit" another transaction thus not delaying the source of the event significantly.
- Recursive implementations may be defined, but will fail to execute correctly. For instance a field trigger function invoked during an insert to file A could attempt to insert data into file B, possibly causing itself to be invoked in a recursive situation, and thus to fail.

[↑ 2.8.3 Trigger Functions](#)

Activate a Trigger Function

To associate a data dictionary field or a file to a trigger function it is necessary to take either the "Review, change or create field rules and triggers" from the field control menu if the trigger is to be created at dictionary level, or the "Review or change file rules and triggers" if the trigger is to be created at the file level.

When the "Add a Trigger" option has been selected, the field must then be associated with a trigger function.

Information can then be entered which will specify when the trigger function is to be activated (for example, before an update of the field is carried out), and if the trigger function should only be activated under certain conditions. See [Field Rules/Triggers](#) for details on entering trigger information.

[↑ 2.8.3 Trigger Functions](#)

Exactly When Are Triggers Invoked?

Before Open / Close

Before open/close triggers are invoked immediately before an attempt is made to open or close a file (or a view of it).

A before open trigger is invoked when a file or a view is opened and or another view has not already opened the file.

This means that if your logic is "Open View 1", then "Open View 2" (where view 1 and view 2 are both based on the same file) then the trigger would be invoked when you open view 1, but not when you open view 2.

A before close trigger is invoked when a file or a view is closed and another view still does not have the file open.

This means that if your logic is "Close View 1", then "Close View 2" (where view 1 and view 2 are both based on the same file) then the trigger would be invoked when you close view 2, but not when you closed view 1.

After Open / Close

Is invoked identically to the "before" options but immediately after a successful attempt has been made to close the file.

Before Read

Is invoked immediately before an attempt is made to read a record from a file. Before input triggers have no access to "information" from the file (because the information has not been input yet) so their use should be considered carefully. Access to the key(s) being used to access the file is not possible in this mode so do not design triggers based on this premise.

After Read

Is invoked after a record has been successfully read from a file and just before the details of the record are passed back to the invoking function. Any virtual field logic has been completed by this stage.

Before Insert

Is invoked immediately before an attempt is made to insert a new record into a file. Please note the following:

- The trigger is run even if the requester uses CHECK_ONLY(*YES)
- The insert may still fail (e.g.: duplicate key error). Before insert triggers should not perform database changes. If database changes are to be done

move the trigger into the "after insert" position instead.

- All virtual logic has been completed when the trigger is invoked.

After Insert

Is invoked immediately after a new record has been inserted into a file. Please note the following:

- The trigger is not run when the requester uses CHECK_ONLY(*YES)
- At the time of invocation all batch control logic has been executed.
- If AUTOCOMMIT is used then the commit is issued before the trigger is invoked.

Before Update

Is invoked immediately before an attempt is made to update an existing record in a file. Please note the following:

- The trigger is run even if the requester uses CHECK_ONLY(*YES)
- The update may still fail (e.g.: duplicate key error). Before update triggers should not perform database changes. If database changes are to be done move the trigger into the "after update" position instead.
- All virtual logic has been completed when the trigger is invoked.

After Update

Is invoked immediately after an existing record has been updated in a file. Please note the following:

- The trigger is not run when the requester uses CHECK_ONLY(*YES)
- At the time of invocation all batch control logic has been executed.
- If AUTOCOMMIT is used then the commit is issued before the trigger is invoked.

Before Delete

Is invoked immediately before an attempt is made to delete an existing record from a file. Please note the following:

- The trigger is run even if the requester uses CHECK_ONLY(*YES)
- The delete may still fail (even though very unlikely). Before delete triggers should not perform database changes. If database changes are to be done move the trigger into the "after delete" position instead.

After Delete

Is invoked immediately after an existing record has been deleted from a file.
Please note the following:

- The trigger is not run when the requester uses CHECK_ONLY(*YES)
- At the time of invocation all batch control logic has been executed.
- If AUTOCOMMIT is used then the commit is issued before the trigger is invoked.

[↑ 2.8.3 Trigger Functions](#)

The TRIG_OPER and TRIG_RETC Variables and TRIG_LIST Working List

When a trigger function is invoked it receives 2 things from the invoker:

TRIG_OPER

TRIG_OPER: is an A(6) field which must be defined in the data dictionary. The content of this field defines what database operation has, or is about to be, performed. Refer to [What Codes Are Passed in TRIG_OPER to the Trigger?](#) for details.

TRIG_LIST

TRIG_LIST: is a 2 entry working list containing 0,1 or 2 entries. The number of entries passed depends upon the database operation being performed. Refer to [How Many Entries Are Passed in the TRIG_LIST?](#) for details.

You must define TRIG_LIST as a working list with 2 entries but you must **not** define any fields within it. The required fields are automatically defined by the RDML compiler.

If your trigger is for field #CUSTNO then the single field #CUSTNO is automatically defined in the list just as if you had typed in:

```
DEF_LIST NAMED(#TRIG_LIST) FIELDS(#CUSTNO)
```

If your trigger was for file Z which contained real fields X, A, T and virtual fields Q and B then the list is automatically defined just as if you had typed in:

```
DEF_LIST NAMED(#TRIG_LIST) FIELDS(#X #A #T #Q #B)
```

This automatic definition ensures that the correct names are used and that you do not have to know or key in the correct names in the correct order.

Remember that the automatic definition is done from the "active" definition of file Z. So if you changed file Z to have X, A, V, T, Q and B as fields and then recompiled the trigger **before** you made the changed file Z "operational" it would automatically define from the unchanged "active" (X, A, T, Q, B) version of file Z.

If you then made the file Z "operational" it would set up its trigger invocations using X, A, V, T, Q, B as the list layout. This is a clear mismatch and would cause unpredictable results.

This mistake would be typified by decimal data errors or by data being "offset" within fields in the list. If this type of problem occurs when a trigger is invoked you should recompile it.

So, when changing a file definition, always make the file "operational" **before** attempting to recompile its associated trigger functions.

To "get" values from the list use the GET_ENTRY command.

This means that when using a trigger for field #CUSTNO you must "get" the correct value of #CUSTNO from the list by using GET_ENTRY NUMBER(?) FROM_LIST(#TRIG_LIST).

Likewise, to get the values of #X through #B in the file Z example you would need also need to "get" them by using GET_ENTRY NUMBER(?) FROM_LIST(#TRIG_LIST).

Only use the list operations SELECTLIST, GET_ENTRY and UPD_ENTRY against the list TRIG_LIST. Only ever issue UPD_ENTRY operations against entry number 1. When a trigger function terminates it returns 2 things to the invoker:

TRIG_RET: is an A(2) field which must be defined in the data dictionary. At the point of return it must be set to "OK", "ER", and in some situations, "VE". See the following sections for more details of the meaning and use of these return codes.

TRIG_LIST: is the 2 entry working list containing 0,1 or 2 entries previously described. You may alter the data in the **first** entry passed by using the UPD_ENTRY command. If you do this in a "before" operation then you will actually alter the data that is inserted or updated into the file.

Likewise if you do this in an "after read" operation you will alter the data that is passed back to the function that issued the read request.

It is strongly recommended that you do **not** use this facility to "communicate" between serially invoked trigger functions.

Only use the list operations SELECTLIST, GET_ENTRY and UPD_ENTRY against the list TRIG_LIST. Only ever issue UPD_ENTRY operations against entry number 1.

[↑ 2.8.3 Trigger Functions](#)

What Codes Are Passed in TRIG_OPER to the Trigger?

When a trigger function is invoked it can access field TRIG_OPER to determine what operation was in progress when it was invoked.

The values passed in TRIG_OPER are:

Operation In Progress Value In TRIG_OPER

Before Open	BEFOPN
After Open	AFTOPN
Before Close	BEFCLS
After Close	AFTCLS
Before Read	BEFRED
After Read	AFTRED
Before Insert	BEFINS
After Insert	AFTINS
Before Update	BEFUPD
After Update	AFTUPD
Before Delete	BEFDLT
After Delete	AFTDLT

[↑ 2.8.3 Trigger Functions](#)

How Many Entries Are Passed in the TRIG_LIST?

When a trigger is invoked it is passed a working list called TRIG_LIST that contains details of the field or file record that is being actioned.

When invoked, TRIG_LIST may contain 0, 1 or 2 entries:

Number of Entries	Meaning / Content
0	There are no details available in this context (e.g.: before open, before read).
1	There is one set of details available in the current context (e.g.: after read).
2	There are two sets of details available in the current context. Entry 1 is the new details and entry 2 is the previous details. (e.g.: before/after update images).

The number of entries passed in the list varies with the operation in progress according to the following table:

Operation In Progress	Number Of Entries In TRIG_LIST
Before Open	0
After Open	0
Before Close	0
After Close	0
Before Read	0
After Read	1
Before Insert	1
After Insert	1
Before Update	2

After Update	2
Before Delete	1
After Delete	1

[↑ 2.8.3 Trigger Functions](#)

What Return Codes Are Used in TRIG_RETc and How Can They Be Set?

Whenever a trigger function completes its completion status is subjected to 3 tests:

1. The return code TRIG_RETc is checked for "OK" (uppercase). If not "OK", then the trigger is deemed to have failed.
2. The function completion status is tested. If not okay, then the trigger is deemed to have failed. A function will give a "bad" completion status if it issues an ABORT command, or hits an ENDCHECK with no last display, or uses an invalid array or substring reference, etc, etc.
3. The IBM i completion status is tested. If not okay, then the trigger is deemed to have failed (e.g.: trigger function not found).

When a trigger function is deemed to have failed, a return code is then issued to the actual invoking function issuing the database operation (i.e. the function doing the SELECT or UPDATE or INSERT) according to the following table:

Operation In Progress Return Code

Before Open	OK, ER
After Open	OK, ER
Before Close	OK, ER
After Close	OK, ER
Before Read	OK, ER
After Read	OK, ER
Before Insert	OK, VE
After Insert	OK, ER
Before Update	OK, VE
After Update	OK, ER
Before Delete	OK, VE
After Delete	OK, ER

You should not return values in TRIG_RET_C outside of those specified in this table.

Nor should the invoking RDML I/O command be set up to do any special "trapping" or "handling" when it knows that there is an underlying trigger. Such an approach would create very complex designs and defeat the whole purpose for which triggers were introduced (i.e. being "invisible" to the upper layer of functionality).

RDML functions doing I/O operations are recommended to not use the IO_ERROR or VAL_ERROR parameters (like any other normal RDML functions).

Leave the default values and let the standard error handling solve the problem. Use a "binary" approach to doing I/O operations - it either worked or it didn't work - and if it didn't work let the standard error handling solve the problem.

Coding your own I/O error traps in RDML functions is not recommended unless they are of a very specialized nature (e.g.: setting up work files). Failing to observe this recommendation will lead to overly complex implementations that exhibit no real business benefit and cost significantly more to develop and maintain.

An "OK" response indicates that the operation completed normally.

An "ER" response sets the IO_ERROR parameter in the RDML command that issued the I/O request.

A "VE" response sets the VAL_ERROR parameter in the RDML command that issued the I/O request.

Note that the "VE" response is **only** possible for before insert, before update and before delete.

This allows triggers in these positions to act like "extended validation checkers". However, a trigger set up as an "extended validation checker" cannot actually "flag" a specific field as being in error.

It can indicate an error has occurred, and it can issue error message details, but it cannot flag the specific field in error in the same way that a normal validation rule can, or in the same way that a normal validation checking function can (see function option *xxx_FIELD_VALIDATE).

However, an "extended validation checker" defined as a trigger has one advantage over a normal validation checking function: it has access to all the

values in the record of a file that is being inserted, deleted or updated - whereas a normal validation function only has access to the individual field value with which it is associated.

Using a trigger as an "extended validation checker" is a very powerful facility, especially when the "before and after images" available to the before update check are considered, and as such it can be very useful at times.

However:

- Reserve the use of "extended validation triggers" for truly complex situations. Do not use this facility without even considering the normal dictionary facilities.
- Where "extended validation check" type triggers are to be used, have just one per file and encapsulate all the rules inside it. You then have just one trigger that supports the action "Validate".
- Make trigger functions exhibit "insulated modularity". Like action bar functions they should exhibit these "OO" like characteristics:
 - They should perform one and only one "action".
 - They should not expect other triggers or virtual code logic to precede, or to follow them.
 - They should operate "standalone".
 - They should be small and robust. When a trigger is invoked to perform an action it should just do that single action or issue an error message indicating why it cannot.

[↑ 2.8.3 Trigger Functions](#)

Triggers - A Classic Example

The following example is a classic example of how a trigger function should be used.

It takes a complex business rule and "encapsulates" it into a trigger.

Next the trigger is linked to the associated database file and the business rule is performed automatically whenever the specified event occurs.

It is a classic example because it clearly demonstrates how triggers can "encapsulate" complex rules and associate them directly with the "object" (i.e. file).

The Business Problem

ACME Engineering run a payroll system.

The Employee Master file (EMPL) contains two fields called "SALARY" and "WEEKPAY".

SALARY is the annual salary that the company has contracted to pay the employee.

WEEKPAY is the amount paid to the employee each week.

WEEKPAY is arrived at via a complex set of rules.

For a new employee the WEEKPAY calculation is relatively simple, but when an employee's SALARY is changed the complex calculation involves both the new SALARY figure and the **previous** SALARY figure.

The Trigger Function

The first step in defining the trigger is to define the trigger function that encapsulates **all** the WEEKPAY rules into one and only one place.

This is a fundamental of good trigger design.

The following function may have been coded to handle this:

```
FUNCTION OPTIONS(*DIRECT *NOMESSAGES *MLOPTIMIZE)
    RCV_LIST(#TRIG_LIST) TRIGGER(*FILE EMPL)
```

```
/* Define the standard trigger list which will contain the */
/* before and after images of the EMPL file record. These */
/* fields are automatically added to the list definition */
/* by the RDML compiler.                                     */
```

```

DEF_LIST NAME(#TRIG_LIST) TYPE(*WORKING) ENTRYS(2)

/* Now examine exactly what event has occurred      */

CASE OF_FIELD(#TRIG_OPER)

/* A new employee is being created */

WHEN VALUE_IS(= BEFINS')
    GET_ENTRY NUMBER(1) FROM_LIST(#TRIG_LIST)
    << calculate correct value into field WEEKPAY >>
    UPD_ENTRY IN_LIST(#TRIG_LIST)

/* An existing salary has been changed */

WHEN VALUE_IS(= BEFUPD')

    DEFINE FIELD(#OLDSALARY) REFFLD(#SALARY)
    GET_ENTRY NUMBER(2) FROM_LIST(#TRIG_LIST)
    CHANGE FIELD(#OLDSALARY) TO(#SALARY)

    GET_ENTRY NUMBER(1) FROM_LIST(#TRIG_LIST)
    << calculate correct value into WEEKPAY >>
    << using OLDSALARY in the calculations >>
    UPD_ENTRY IN_LIST(#TRIG_LIST)

OTHERWISE
    ABORT MSGTXT('WEEKPAY trigger function invalidly invoked')

ENDCASE

CHANGE FIELD(#TRIG_RETC) TO(OK)
RETURN

```

Activating the Trigger Function

Now that the trigger function has been defined it needs to be activated. To do this, access the definition of file EMPL and associate two trigger invocation events with it.

The first would be specified as "BEFORE INSERT" and would not have any associated conditions. This means that the trigger function will be called whenever an attempt is made to create a new employee.

The second would be specified as "BEFORE UPDATE" and would have an associated condition which would look something like this:

SALARY NEP SALARY

i.e. salary is not equal to previous salary

which says that the trigger should be activated "BEFORE UPDATE" but only if the employee's SALARY has changed.

Defining the "BEFORE UPDATE" event like this is very efficient because it means that the trigger will not be activated when the employee's salary has not been changed (which will probably be most of the time).

If WEEKPAY had to be recalculated when the SALARY changed or when the COMPANY that the employee worked for changed, then you would define the invocation event like this instead:

SALARY NEP SALARY OR COMPANY NEP COMPANY

i.e. salary is not equal to previous salary

or company is not equal to previous company

If WEEKPAY was to always be recalculated, then you would not have to define two separate invocation events. You could simply define one event (with no conditions) and indicate that the trigger should be invoked "BEFORE INSERT" and "BEFORE UPDATE".

Of course, this means that every single insert or update of an employee would cause the trigger function to be invoked.

Key Things to Note About this Example

This example demonstrates some of the key elements of good trigger design and use:

- The "encapsulation" principle. The WEEKPAY calculation "method" is "encapsulated" in **one and only one** function. If it has to be changed it only has to be changed in one place.
- Deferment. The existence of the WEEKPAY method does not have to be defined, or even known about, during initial system design.

This also means that a "method" can be introduced into an application design at any time. For instance, the WEEKPAY method does not have to be defined

before any applications that create or update employees are. The create and update applications can be defined and tested first. When the WEEKPAY method is created and defined it will immediately begin to affect the processing of all existing applications.

- Reusability. The WEEKPAY calculation method is automatically and implicitly reused by any application that creates or changes employee details. The trigger could be activated from a normal NPT device via an "Employee Maintenance" function, or from a PC application via the LANSAs Open facility.
- Transparency. The fact that the WEEKPAY logic is present and being used is invisible and probably immaterial to an RDML builder creating an "Employee Maintenance" function.
- Separation of the "method" from the "event". The trigger function defines what to do when an "event" happens (i.e. the "method").

However, it does **not** have to detect the occurrence of the event.

For example, the function defined previously defines a "method" called "Calculate Weekly Pay".

The business rules says that weekly pay must be (re)calculated when a new employee is taken on, or when an existing employee's salary is changed, or when an existing employee moves to another company.

The actual "event" is defined in the LANSAs data dictionary.

[↑ 2.8.3 Trigger Functions](#)

Examples of Trigger use

Example 1: Calculates the current balance of the account:

Object : Account (ACNT)

Trigger Field : "Account Balance" (ACCBAL)

Trigger Method : Calculates the current balance of the account.

Trigger Event(s) : 1. AFTER READ when ACCBAL is REF (referenced)

Invoked By : (i.e. How is an account balance retrieved ?)

```
        FETCH FIELDS(...#ACCBAL....)
        FROM_FILE(ACNT)
or SELECT FIELDS(...#ACCBAL....)
        FROM_FILE(ACNT)
```

Comments : Quite efficient because ACCBAL is only calculated when the requester asks for it.

Example 2: Submits batch order print job:

Object : Order (ORDR)

Trigger Field : "Print Required" (PRINT_REQ)

Trigger Method : Submits batch order print job. Batch function prints order and updates PRINT_REQ to 'N'.

Trigger Event(s) : 1. AFTER INSERT when PRINT_REQ EQ 'Y'

```
        2. AFTER UPDATE when PRINT_REQ EQ 'Y'
           and PRINT_REQ NEP PRINT_REQ
```

Invoked By : (i.e. How is an order submitted for printing ?)

```
CHANGE #PRINT_REQ 'Y'  
UPDATE FIELD(#PRINT_REQ) IN_FILE(ORDR)
```

Comments : It may seem a little obscure as an IBM i based trigger but what if the update comes from a PC based application via the LCOE facility, or from a batch job that starts automatically every morning and automatically selects certain orders for printing ?

[↑ 2.8.3 Trigger Functions](#)

Triggers - Restrictions and Limitations

Trigger logic on associated batch control files is not performed. If file A uses file B as a batch control header file, then triggers associated with B are not invoked when I/O operations to A need to insert/update file B "batch header" records.

[↑ 2.8.3 Trigger Functions](#)

Triggers - Some Do's and Don'ts

Some Do's

- Do experiment with small test cases using triggers so that you are comfortable with what they are and how they work before attempting to implement a complex application involving triggers.
- Do remember that when you change the type or length of a field in the data dictionary (that has associated triggers) you should recompile:
 - All trigger functions associated with the field.
 - All I/O modules of files that contain the field as a real or virtual field.
 - All functions that make *DBOPTIMIZE references to file(s) containing the field.
- The list of objects to recompile is easily obtained by producing a full listing of the definition of the field.
- Remember that when you change the layout of a database file (that has associated triggers) you should recompile:
 - The I/O module of file.
 - All trigger functions associated with the file.
 - Any functions that make *DBOPTIMIZE references to the file.

The list of objects to recompile is easily obtained by producing a full listing of the definition of the file.

Some Don'ts

- Do not do any I/O to the file with which the trigger is linked. Attempting such I/O directly, or indirectly, may cause a recursive call to the file I/O module. Do not attempt to use *DBOPTIMIZE to circumvent this rule. Such attempts will cause the file cursor of the active I/O module to become lost or corrupted.
- Do not use triggers on files that have more than 799 real and virtual fields (the 800th field position is reserved for the standard @@UPID field).
- Do not make triggers too expensive to execute. For example, an unconditioned trigger that is always executed after reading from a file doing, say, 3 database accesses, will at least quadruple the time required to read the base file. Triggers are a very useful facility but they are **not** magic. When you set up a trigger to do a lot of work, then your throughput will be reduced

accordingly. The use of triggers and the estimation of the impact that they exert on application throughput is entirely your responsibility as an application designer.

- Do not introduce dependencies between triggers. For instance, trigger A (before update) sets a value in field X, say. Setting up trigger B (also before update), to run after trigger A, with the "knowledge" that trigger A has been executed first (and thus set field X) is not a good idea. This is an example of "interdependence" between triggers and it is **not** a good way to use triggers. In this case the logic in trigger B should be inserted directly into trigger A following the point that it sets a value into field X.
- Do not use ABORT when a user exit is called from a Trigger function. When ABORT is issued in the Trigger Function, the I/O Module is able to intercept the ABORT and passes a Trigger error status back to the Function. However, when the ABORT is issued in the (user exit) Function, called by the Trigger, the ABORT is interpreted in the standard way because the Function is not aware that the call was from a Trigger and it does not make any difference. Using ABORT in these situations (e.g. validations) is not recommended.
- It is **very strongly recommended** that you do not design triggers in such a way as that "normal" RDML functions doing I/O operations are "aware" of their existence, and attempt to **directly** "communicate" with them in any way (e.g.: *LDA, data areas, etc).

Where trigger "requests" are to be supported, introduce a virtual (or real) field into the file definition and use it to "fire" the trigger in the normal way.

[↑ 2.8.3 Trigger Functions](#)

3. Files

- 3.1 File Definition
- 3.2 Real Fields in File
- 3.3 Virtual Fields in File
- 3.4 Logical Views
- 3.5 Access Routes
- 3.6 File Attributes
- 3.8 Batch Control
- 3.9 Virtual Derivation
- 3.10 File Rules and Triggers
- 3.11 File Compile Options
- 3.12 Load Other File

Also See

[Editing Files](#) in the *User Guide*.

[Developing Databases](#) in the *Developer Guide*.

3.1 File Definition

[3.1.5 File Description](#) [3.1.4 File Library](#) [3.1.1 File Type](#)
[3.1.3 File Identifier](#) [3.1.2 File Name](#)

Also See

[Developing Databases](#) in the *Developer Guide*.

[↑ 3. Files](#)

3.1.1 File Type

File Type defines whether the file is created and maintained by LANSAs or is an externally-defined file which is maintained by another system.

You can easily identify the file type by the icon used or you can look at the File Description in the File Attributes tab.

-  A LANSAs file, created and maintained by LANSAs.
-  An IBM i Other file. Files with this icon are IBM i files created and maintained by another application. The definition of the file has been loaded into LANSAs from the IBM i via Load Other File.
-  A PC Other file. Files with this icon are non-IBM i files that are created and maintained by another application. The definition of the file has been loaded into LANSAs from an ODBC data source. For more information refer to [PC Other Files](#) in the *Visual LANSAs Developer's Guide*.

[↑ 3.1 File Definition](#)

3.1.2 File Name

Mandatory.

Specify the name by which the file is to be known.

Rules

- Must be a valid [LANSA object name](#).

Platform Considerations

- Refer to [LANSA object name](#).

Tips & Techniques

- Refer to [LANSA object name](#).

Also See

[3.1.4 File Library](#)

[3.1.3 File Identifier](#)

[↑ 3.1 File Definition](#)

3.1.3 File Identifier

Mandatory.

Specify the identifier by which the file is to be known.

Rules

- Must be a valid [LANSA object name](#).
- Combination of file identifier and [3.1.4 File Library](#) must be unique.

Platform Considerations

- IBM i: If using native RPG programming language, a maximum of 8 character names should be used.

Tips & Techniques

- The use of the same file identifier in different libraries is possible using LANSAs. However it is strongly recommended that this procedure is NOT used. Instead it is better to install multiple LANSAs systems to service different environments such as development and production.

Also See

[3.1.2 File Name](#)

[3.1.4 File Library](#)

[↑ 3.1 File Definition](#)

3.1.4 File Library

Mandatory.

Specify the library in which the new file is to reside.

Default=Partition default file library name.

Rules

- Library name must conform to object naming standards for the operating system.
- When the library name is combined with the [3.1.3 File Identifier](#), it must form a unique name for the file.
- This field is pre-filled with the partition default file library name. LANSAs does not check whether the library specified exists or not.

Warnings

- The library specified must exist and you must be authorized to use it. LANSAs does not check whether the library specified exists or not.

Platform Considerations

- IBM i: Avoid creating files that reside in your IBM i "current" library.

Tips & Techniques

- The use of the same [3.1.3 File Identifier](#) in different libraries is possible using LANSAs. However it is strongly recommended that this procedure is NOT used. Instead it is better to install multiple LANSAs systems to service different environments such as development and production.

Also See

[3.1.3 File Identifier](#)

[↑ 3.1 File Definition](#)

3.1.5 File Description

Mandatory.

Specify the description to be associated with the file. This description is used within LANSAs and within the operating system to aid system users in identifying the file. If the partition is multilingual, the description specified for the default partition language will be used as the default for other languages. File descriptions for other languages can be updated using [3.6 File Attributes](#).

Rules

- Maximum length is 40 characters.

Platform Considerations

- IBM i: Use of CUA standards for identification are recommended.

Tips & Techniques

- Choose the file description carefully as it is used more often than the [3.1.3 File Identifier](#). For example, reporting tools like LANSAs Client will display File Descriptions for end-users to select files.

[↑ 3.1 File Definition](#)

3.2 Real Fields in File

The real fields in the file are the normal fields found in any table/file definition and are assembled together to form the record format of the associated table/file. Such fields actually exist in the database table/file and their content (or value) can be extracted from any record in the table /file at any time.

When adding a new real field to a file, refer to:

[3.2.1 Field Name](#)

[3.2.3 Field Key Position](#)

[3.2.2 Field Sequence](#)

[3.2.4 Virtual Field Flag](#)

Also See

[3.3 Virtual Fields in File](#)

[Developing Databases](#) in the *Developer Guide*.

[Fields in File tab](#) in the *User Guide*.

[↑ 3. Files](#)

3.2.1 Field Name

Mandatory.

Specify the name of a field that is to be included in the file definition.

Rules

- A field used in a file definition must be defined in the LANSAs Repository.
- **Note:** If the "Multi-add fields" interface is being used to add fields to the file, new fields may be created in the repository. Refer to [Field Name Definition](#).

Tips & Techniques

- LANSAs automatically includes one field into every file it creates. The field is called [3.7 The @@UPID Field in LANSAs Created Files](#) and is used by the LANSAs database I/O modules to check for 'crossed updates'. Although this field is defined in the LANSAs Repository it should NEVER be manually included into a file definition. The field does not appear in the list of fields in the file but will appear in the physical database definition.
- The allowed field types in a file are determined by the partition settings. For more details, refer to [RDML and RDMLX Partition Concepts](#) in the *Administrator Guide*.

Also See

[3.2.3 Field Key Position](#)

[3.2.2 Field Sequence](#)

[↑ 3.2 Real Fields in File](#)

3.2.2 Field Sequence

Mandatory.

Specify the position of the field relative to other fields in the file.

Field sequence is used to determine the physical order of the fields in the file, or the order of the columns in the table.

Rules

- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a field is added. Sequence number is updated when the order of the fields is updated in the list of fields.

Tips & Techniques

- Field sequence is not related to [3.2.3 Field Key Position](#) . However, the key fields are typically the first fields listed in the file.
- Field sequence is important when templates are used to generate RDML code. By default, fields are listed in the physical order in the file. Generally, the most important fields will be grouped as the first fields in the file.

[↑ 3.2 Real Fields in File](#)

3.2.3 Field Key Position

Specify if the field is assigned a key position so that it is included to form the primary key of the file.

Fields that form the primary key of the file should be listed first key in hierarchy (most significant key) to last key in the hierarchy (least significant key).

Rules

- The primary key of a file (which is composed of all the fields in the key concatenated in the order specified) must be UNIQUE.
- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a key field/value is added. Sequence number is updated when the order of the key is updated in the list of keys.

Platform Considerations

- IBM i: No more than one record in a file can have any given primary key. This rule is enforced by features in the operating system (DDS keyword UNIQUE) and can never be violated. Attempting to add a record to a file with the same primary key as a record already in the file will result in a "duplicate key error".

Tips & Techniques

- There is no requirement to specify a primary key when setting up a file definition, but it is strongly recommended that each file be set up with a key.
- LANSAs automatically handles duplicate key errors and there is no need for user logic to handle or check for them.
- The existence of a unique primary key is important to LANSAs because it uniquely identifies one and only one record in a file.
- Field key position is not related to [3.2.2 Field Sequence](#). However, the key fields are typically the first fields listed in the file.

Also See

[3.2.2 Field Sequence](#)

[↑ 3.2 Real Fields in File](#)

3.2.4 Virtual Field Flag

Specify whether this field will be a virtual field in the file definition.

When adding multiple fields to a file, you may specify that the field will have a virtual field definition associated with it.

Default=No (unchecked/not selected)

[↑ 3.3 Virtual Fields in File](#)

3.3 Virtual Fields in File

Virtual fields are fields that do not actually exist in the database file, but are dynamically derived from "real" fields in the file.

[3.3.1 Virtual Field Definition](#)

[3.3.2 Date Conversion](#)

[3.3.3 Substring](#)

[3.3.4 Concatenation](#)

[3.3.5 Mathematical Calculations](#)

[3.3.6 Code Fragment](#)

Also See

[Virtual Field Development](#) in the *Developer Guide*.

[Fields in File Tab](#) in the *User Guide*.

[3.5.2 Predetermined Join Field Definitions](#)

[3.2 Real Fields in File](#)

[3.9 Virtual Derivation](#)

[↑ 3. Files](#)

3.3.1 Virtual Field Definition

[Virtual Field Name](#)

[Virtual Field Type](#)

[Virtual Field Sequence](#)

[Derive value when record is read](#)

[Populate real field when writing to file](#)

Also See

[Virtual Field Development](#) in the *Developer Guide*.

[↑ 3.3 Virtual Fields in File](#)

Virtual Field Name

Mandatory.

Specify the name of a field that is to be included in the file definition as a virtual field.

Rules

- A field used in a file definition must be defined in the LANSAs Repository.
- The field type must be in agreement with the [Virtual Field Type](#) selected. For example, a numeric field type must be specified for a mathematical calculation.
- The field length is validated for substring and concatenation virtuals.

Tips & Techniques

- Follow the field naming standards when using virtual fields. For example, name all virtual fields with a 3 character suffix of "VIR". This will help developers identify a field as a virtual field.

Also See

[3.2 Real Fields in File](#)

[↑ 3.3 Virtual Fields in File](#)

Virtual Field Type

Mandatory.

Specify the type of derivation to be used for the virtual field.

Rules

Allowable values are:

[3.3.2 Date Conversion](#) Map a date held in a real field in format (YYMMDD) into a virtual field in format (DDMMYY).

[3.3.3 Substring](#) Substring 1 real field into multiple virtual fields

[3.3.4 Concatenation](#) Substring multiple real fields into one virtual field

[3.3.5 Mathematical Calculations](#) Specify the mathematical calculation extended definition of the virtual field currently being worked with.

[3.3.6 Code Fragment](#) Allows you to specify RDMLX code to populate a virtual field when reading from a file, and to populate a real field when writing to a file.

Tips & Techniques

- The [Virtual Field Name](#)'s type and length must be in agreement with the type of virtual field selected. For example, a numeric field type must be specified for a mathematical calculation. The field length is validated for only some types of virtuals. Check that the length of the field is correct for the virtual type selected.

Also See

[Virtual Field Name](#)

[↑ 3.3 Virtual Fields in File](#)

Virtual Field Sequence

Mandatory.

Specify the sequence/order of the virtual field relative to the other virtual fields in the file.

Rules

- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a virtual field is added. Sequence number is updated when the order of the fields is updated in the list of fields.

Tips & Techniques

- The ordering of the virtual fields can be resequenced by varying the value in this field.
- The sequence number for virtual fields only applies within the group they are listed in.

[↑ 3.3 Virtual Fields in File](#)

Derive value when record is read

Specify whether the virtual field should be derived from the file after reading the real field.

This option means that the virtual field is created when you read the information from the file. The information will now be derived and available for display. For example, a date is stored as format YYYY/MM/DD so the file can be properly sorted by date, but the information is displayed to the user as DD/MM/YY.

Default=NO (unchecked).

Rules

- At least one of [Populate real field when writing to file](#) and [Derive value when record is read](#) must be selected. Both values may be selected.

Tips & Techniques

- Virtual field derivation is also performed relative to predetermined joined fields. Refer to [3.5.1 Access Route Definitions](#) and [Derivation](#).

Also See

[Populate real field when writing to file](#)

[↑ 3.3 Virtual Fields in File](#)

Populate real field when writing to file

Specify whether the "real" field should be setup from the "virtual" field before writing to the file.

This option means that information entered in a virtual field will be written out to the file field(s) used to define the virtual fields. The virtual fields are the user input fields and will write information to a real field in the file. For example, the user enters a date in DD/MM/YY and it is then converted and stored in the file as YY/MM/DD.

Default=NO (unchecked).

Rules

- At least one of [Populate real field when writing to file](#) and [Derive value when record is read](#) must be selected. Both values may be selected.

Tips & Techniques

- This option is not supported for the [3.3.5 Mathematical Calculations](#) virtual.
- You can add validation rules to virtual fields but be careful that the virtual field is included on the display so that it can be highlighted when an error occurs. In the case of a date virtual, you would only display the virtual field with the validation rule and not the real field on the screen.
- Virtual field derivation is also performed relative to predetermined joined fields. Refer to [3.5.1 Access Route Definitions](#) and [Derivation](#).

Also See

[Derive value when record is read](#)

[↑ 3.3 Virtual Fields in File](#)

3.3.2 Date Conversion

The date virtual will convert the format of an existing date field in the file. For example, when a date is held in a real field in format YYMMDD it can be easily mapped into a virtual field in format DDMMYY. The real field may be the most commonly used format for ordering the file, but the virtual field format may be the most commonly used for display or printing.

Refer to:

[Source Field](#)

[Source Format](#)

[Target Format](#)

Also See

[Virtual Field Type](#)

[↑ 3.3 Virtual Fields in File](#)

Source Field

Mandatory.

Specify the field (real or virtual) in the physical file to be used for the date conversion to the resulting virtual field nominated by the [Virtual Field Name](#).

Rules

- If the virtual field is defined as [Populate real field when writing to file](#), the source field must be big enough to contain the result of/value for the date conversion, otherwise unpredictable results could occur.

Also See

[↑ 3.3.2 Date Conversion](#)

Source Format

Mandatory.

Specify the date format which is currently being used by the [Source Field](#) in the file.

Rules

Valid date format values are:

SYSFMT	QDATFMT
DDMMYY	Day, Month and Year
MMDDYY	Month, Day and Year
YYMMDD	Year, Month and Day
DDMMYYYY	Day, Month, Century and Year
MMDDYYYY	Month, Day, Century and Year
YYYYMMDD	Century, Year, Month and Day
YYMM	Year and Month
MMYY	Month and Year
MMYYYY	Month, Century and Year
YYYYMM	Century, Year and Month
SYSFMT8	QDATFMT including Century
CYYMMDD	Century indicator, Year, Month and Day

Note:

The CYYMMDD format is only valid for a real field in LANSAs for the IBM i in virtual field derivation. It can be defined in Visual LANSAs, but not made operational.

Also See

[Target Format](#)

[↑ 3.3.2 Date Conversion](#)

Target Format

Mandatory.

Specify the desired date format to be used by the virtual field nominated in the [Virtual Field Name](#).

This is the date format that the virtual field is converting to.

Rules

Valid date format values are:

SYSFMT	QDATFMT
DDMMYY	Day, Month and Year
MMDDYY	Month, Day and Year
YYMMDD	Year, Month and Day
DDMMYYYY	Day, Month, Century and Year
MMDDYYYY	Month, Day, Century and Year
YYYYMMDD	Century, Year, Month and Day
YYMM	Year and Month
MMYY	Month and Year
MMYYYY	Month, Century and Year
YYYYMM	Century, Year and Month
SYSFMT8	QDATFMT including Century

Warning

- If an alphanumeric date field (real or virtual) is being converted to a resulting numeric date field (real or virtual), a check will be automatically performed to ensure the resulting numeric date field contains all numeric characters (0 to 9). If the resulting numeric date field does not contain all numeric characters (i.e. alpha characters have been found) the resulting numeric date field will be set to 0.

Also See

[Source Format](#)

[↑ 3.3.2 Date Conversion](#)

3.3.3 Substring

The substring virtual field allows you to access either part of a field or a complete file record. For a substring virtual, you will need to specify the field from which you want to substring. A substring virtual can be based on:

- a real field in the file
- another virtual field in the file
- the complete record contents of the file (indicated by *RECORD).

Refer to:

[Substring from Field](#)

[Start Position](#)

Also See

[Virtual Field Type](#)

[↑ 3.3 Virtual Fields in File](#)

Substring from Field

Mandatory.

Specify the source field (real or virtual) in the physical file or the record contents of the physical file (*RECORD) that is to be used for the substring operation.

Rules

Allowable values are:

- A real or virtual field defined in the file definition.
- *RECORD indicates that the full record content of the file is to be used.
- You cannot perform a substring from a packed numeric field to an alpha field.

Warnings

- If *RECORD has been specified for this field, the utmost care should be taken in the substring of the file record. There will be no validation checks to ensure data type/length compatibility. Virtual fields formed as a result of a substring operation specifying *RECORD are totally your responsibility.
- PLEASE be careful when performing this option when using *RECORD as field definition errors could cause unpredictable results.

Tips & Techniques

Following are special notes for substring virtuals when used with alpha field into a numeric field:

- The field should only contain the digits 0-9. Any other character, including a sign character ('+' or '-'), will give unpredictable results.
- Substring is performed from left to right, therefore if a field containing '123.45' is substringed into position 1 of a signed (6,2) field (which is initially set to *ZERO), the value will be set to 1234.50.

Also See

[Start Position](#)

[↑ 3.3.3 Substring](#)

Start Position

Specify the start position within the [Substring from Field](#).

There is no requirement for an end position as this is automatically calculated by adding the length of the virtual field to the start position. (Refer to [Virtual Field Name](#).)

Mandatory. Default=1

Rules

- Value must be less than the length of the [Substring from Field](#).
- The length of the [Substring from Field](#) minus the value of the Start Position must not be greater than the virtual field nominated in the [Virtual Field Name](#) (unless *RECORD has been specified), i.e. the length of the substring data must fit into the virtual field.

Also See

[Substring from Field](#)

[↑ 3.3.3 Substring](#)

3.3.4 Concatenation

The string concatenation virtual will join multiple fields in the file together and place the result into the defined virtual.

Refer to:

[Concatenated Field Names](#)

[Concatenation Example](#)

[↑ 3.3 Virtual Fields in File](#)

Concatenated Field Names

Mandatory.

Specify each field (real or virtual) in the physical file to be joined together to form the virtual field nominated in the [Virtual Field Name](#).

When the fields have been concatenated, they will form the virtual field starting at position 1.

Rules

- One or more fields must be identified.
- Fields may be real fields, virtual fields or predetermined join fields from the file definition.

Tips & Techniques

- The field defined in the [Virtual Field Name](#) must be of sufficient length to hold the concatenated fields.
- If you are using Predetermined Join Fields to create the virtual field, you must specify that the PJF derivation is **Before virtual fields** in the [Derivation](#) of the [3.5.1 Access Route Definitions](#).

Also See

[Concatenation Example](#)

[↑ 3.3.4 Concatenation](#)

Concatenation Example

Following is an example of a concatenation virtual field.

The Employee file contains the employee's last name in a field SURNAME defined as A(20).

The Employee file contains the employee's first name in a field GIVENAME defined as A(20).

A concatenation virtual field can be created named FULLNAME defined as A(40). This field will have the first name followed by the last name:

Virtual Field Name: FULLNAME

Field Names Length Description

GIVENAME	20	Employee Given Name
SURNAME	20	Employee Surname

Also see

[3.3.6 Code Fragment](#)

[↑ 3.3.4 Concatenation](#)

3.3.5 Mathematical Calculations

The mathematical calculation virtual will perform a set of numeric calculations to determine field. This type of virtual can only be derived when read from the file.

Refer to:

[Factor 1](#)

[Operator](#)

[Factor 2](#)

[Result](#)

[Mathematical Calculation Example](#)

Also See

[Virtual Field Type](#)

[↑ 3.3 Virtual Fields in File](#)

Factor 1

Mandatory.

Specify the first field or value (Factor1) that will be used in a mathematical expression of:

Factor1 (Operator) Factor2 = Result

Rules

- The field (real or virtual) must be defined in the physical file, or a valid numeric literal, or an *WORKnnnnn field.
- If a field name has been specified, its data type must be numeric.
- This is an optional entry for all operation codes, except for when the operation code of "S" is specified. In this case, the Factor1 field must be blank.

Also See

[Mathematical Calculation Example](#)

[↑ 3.3.5 Mathematical Calculations](#)

Operator

Mandatory.

Specify the operator that will be used in a mathematical expression of:

Factor1 (Operator) Factor2 = Result

Rules

The list of operators are:

+ Add

- Subtract

/ Divide

* Multiply

S Set

Warning

- If the operation for a calculation line is / (Divide), a check will be automatically performed to ensure Divide by zero errors are prevented.
- If the value within Value 2 is 0, the Result value will be automatically set to 0. If the value within Value 2 is not 0, then the / (Divide) operation will be performed.

Also See

[Mathematical Calculation Example](#)

[↑ 3.3.5 Mathematical Calculations](#)

Factor 2

Mandatory.

Specify the second field or value (Factor2) that will be used in a mathematical expression of:

Factor1 (Operator) Factor2 = Result

Rules

- The field (real or virtual) must be defined in the physical file, or a valid numeric literal, or an *WORKnnnnn field.
- If a field name has been specified, its data type must be numeric.
- This is a mandatory entry for all operation codes, except for when the operation code of S is specified. In this case the Value 2 is optional. If the operation code is S and Value 2 is blank, the system will automatically set Value 2 field to 0.

Also See

[Mathematical Calculation Example](#)

[↑ 3.3.5 Mathematical Calculations](#)

Result

Mandatory.

Specify the result virtual field or a *WORKnnnnn field that will be used in a mathematical expression of:

Factor1 (Operator) Factor2 = Result

Rules

- The final result in the calculation must be the defined virtual field.

Tips & Techniques

- The *WORKnnnnn fields are exclusively reserved for use in "virtual" code extended definition "mathematical calculations" only.
- The *WORKnnnnn fields do not have to be defined within LANSAs, the *WORKnnnnn field length will be automatically assumed to 30, 9 if the machine is an IBM i.
- All *WORKnnnnn fields that are used in a mathematical calculation are automatically initialized before the first time they are used.
- Multiple *WORKnnnnn fields can be used in mathematical calculations simply by replacing the nnnnn portion of the work field name with a unique replacement value.
- If *WORKnnnnn fields with the same name are used in other virtual fields' mathematical calculations, their values will not be carried forward to the next virtual field's mathematical calculation.

Platform Considerations

- **IBM i:** The *WORKnnnnn fields do not have to be defined within LANSAs, the *WORKnnnnn field length will be automatically assumed to 30, 9 if the machine is an IBM i.
- **Other platforms:** Virtual field calculations support a maximum of 15 digits precision. This means that if the contents of the field in Factor 1, Factor 2 are more than 15 digits, or the result would exceed 15 digits, this will result in an error and/or an incorrect result. To avoid this, if the file is RDMLX, use an RDMLX code fragment. If the file is RDML, use a trigger function to calculate the result field. RDML mathematics will work correctly in a trigger function on all platforms.

Also See

Mathematical Calculation Example

↑ 3.3.5 Mathematical Calculations

Mathematical Calculation Example

Following is an example of a mathematical calculation virtual field.

Calculate the Gross sales price of a transaction by multiplying the Net sales price by the Current sales tax rate and adding the result to the Gross sales price:

Factor1	operator	Factor2	Result
	S	O	GRSPRC
NETPRC	/	100	*WORKNET01
*WORKNET01 *		SALTAX	*WORKTAX01
NETPRC	+	*WORKTAX01	GRSPRC

[↑ 3.3.5 Mathematical Calculations](#)

3.3.6 Code Fragment

The code fragment virtual field allows you to specify RDMLX code to populate a virtual field when reading from a file, and to populate a real field when writing to a file.

A code fragment does not support the full range of RDML/X commands. In Version 11, you are limited to constructs such as If, Case, Dountil, Dowhile, Change, Assign.

At execution time, the code fragment will have read-only access to all fields in the file. You may use trigger functions for complex coding of virtual fields or to update fields in a file.

Code Fragments are specified by selecting *Code Fragments* in the *Virtual field type* dropdown list in the *Details* tab. In the *Details* tab, when you select the *Derive Value when record is read* option, the associated tab for *Virtual Field Derivation* is displayed. Enter the code fragment to be used to derive the virtual field. The code fragment is typically derived from one or more real fields on the file but may also use system variables, multilingual variables and work fields.

When you select the *Populate real field when writing to the file* option, the associated tab for *Real Field Derivation* is displayed. Enter the code fragment to be used to derive the real field values. The code fragment is typically derived from the current virtual field.

In the following example, only the first option is selected, so only the *Virtual Field Derivation* tab is shown:

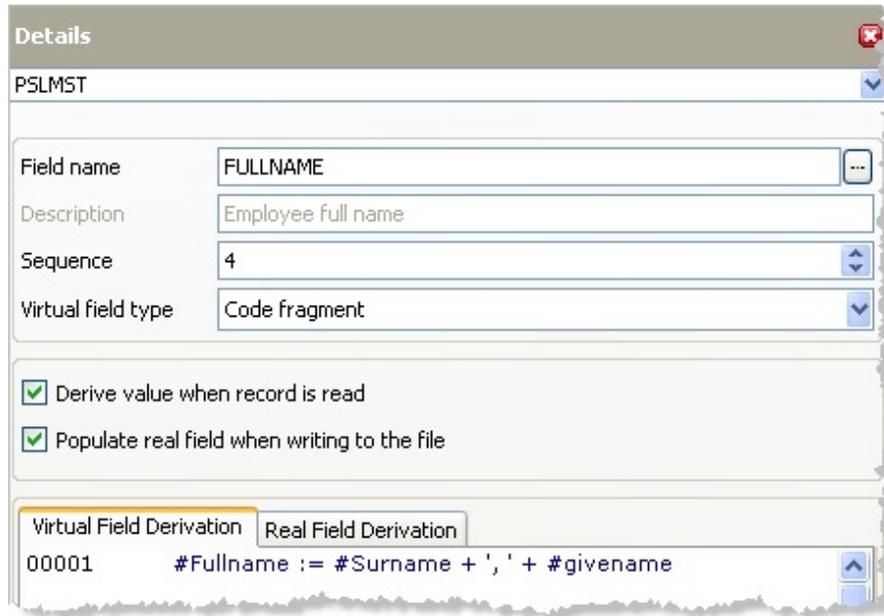
The screenshot shows a 'Details' dialog box for a virtual field. The field name is 'FULLNAME' and its description is 'Employee full name'. The sequence is set to 4, and the virtual field type is 'Code fragment'. Two options are present: 'Derive value when record is read' (checked) and 'Populate real field when writing to the file' (unchecked). The 'Virtual Field Derivation' tab is active, showing the RDMLX code: '00001 #Fullname := #Surname + ', ' + #givenname'.

Field name	FULLNAME
Description	Employee full name
Sequence	4
Virtual field type	Code fragment

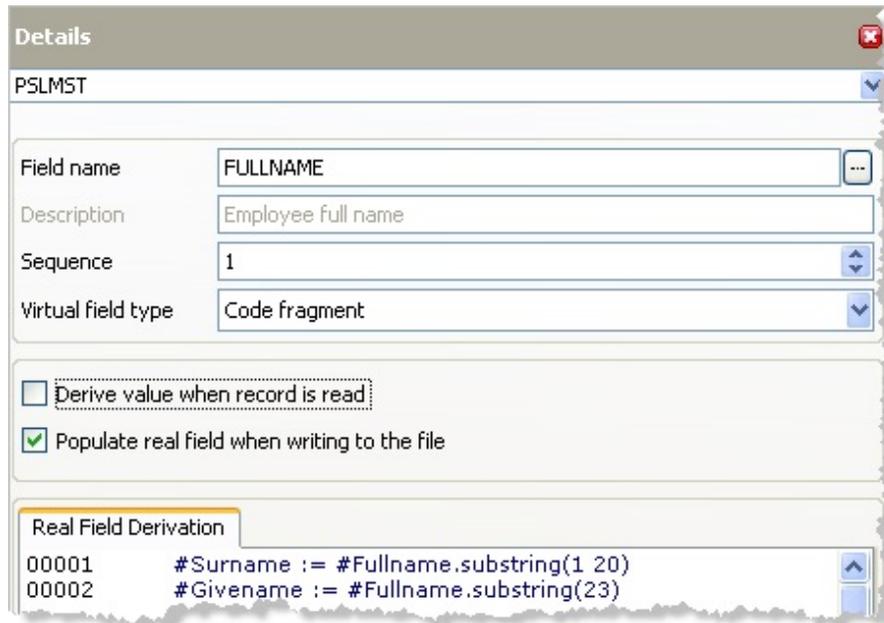
Derive value when record is read
 Populate real field when writing to the file

Virtual Field Derivation
00001 #Fullname := #Surname + ', ' + #givenname

When both options are selected both tabs appear:



When only the second option is selected, only the *Real Field Derivation* tab is shown:



Code fragment vs concatenation

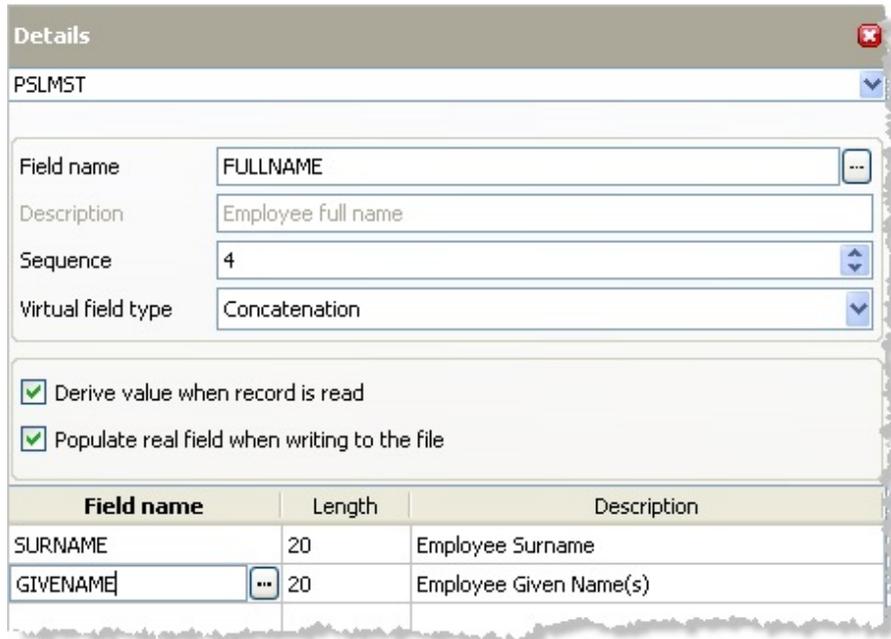
Code fragments give you more control and variability than concatenation. The following example of Concatenation uses the same information to derive the real and the virtual field (simply by applying reverse of definition) and gives you a similar but not quite the same result.

The **code fragment**:

```
#fullname := #surname + ', ' + #givenname
```

would result in Turner , Scott

Concatenation of Surname and Full name would give TurnerScott



The screenshot shows a 'Details' dialog box for a virtual field named 'FULLNAME'. The dialog is titled 'Details' and has a dropdown menu showing 'PSLMST'. The field name is 'FULLNAME', the description is 'Employee full name', the sequence is '4', and the virtual field type is 'Concatenation'. There are two checked checkboxes: 'Derive value when record is read' and 'Populate real field when writing to the file'. Below the form fields is a table with three columns: 'Field name', 'Length', and 'Description'.

Field name	Length	Description
SURNAME	20	Employee Surname
GIVENAME	20	Employee Given Name(s)

Also see

[Code Fragment Virtual Field Derivation](#)

[Code Fragment Real Field Derivation](#)

[3.3.4 Concatenation](#)

[↑ 3.3 Virtual Fields in File](#)

Code Fragment Virtual Field Derivation

A code fragment does not support the full range of RDML/X commands. In Version 11, the user is limited to constructs such as If, Case, Dountil, Dowhile, Change, Assign.

At execution time, the code fragment will have read only access to all fields in the file.

In the following example, the PSLMST file has two fields SURNAME and GIVENAME. SURNAME is A(20) and GIVENAME is A(20). If the SURNAME was "Smith" and GIVENAME was "John", a simple concatenation virtual would create a virtual field called FULLNAME with a result of "Smith John".

Using the following code fragment:

```
#FULLNAME := #SURNAME.Trim + ',' + #GIVENAME.Trim
```

the code fragment virtual field FULLNAME will have a result of "Smith, John".

[↑ 3.3.6 Code Fragment](#)

Code Fragment Real Field Derivation

A code fragment does not support the full range of RDML/X commands. In Version 11, the user is limited to constructs such as If, Case, Dountil, Dowhile, Change, Assign.

At execution time, the code fragment will have read-only access to all fields in the file.

[↑ 3.3.6 Code Fragment](#)

3.4 Logical Views

Logical files or views are created as part of a file definition. They are not stored as separate objects in the repository. Logical views are used to create alternate ways of organizing the data in your files.

[3.4.1 Logical View Definition](#)

[3.4.2 Logical View Keys](#)

[3.4.3 Select/Omit Criterion](#)

Also See

[Logical View Development](#) in the *Developer Guide*.

[Logical Views Tab](#) in the *User Guide*.

[↑ 3. Files](#)

3.4.1 Logical View Definition

[Logical View Name](#)

[Logical View Description](#)

[Unique Key](#)

[Access Path](#)

[Dynamic Select](#)

[Alt. Seq.](#)

[Record Format](#)

[CRTLF/CHGLF Parameters](#)

Also See

[Logical View Development](#) in the *Developer Guide*.

[Logical Views Tab](#) in the *User Guide*

[↑ 3.4 Logical Views](#)

Logical View Name

Mandatory.

Specify the name that is to be assigned to the logical view/ file.

Rules

- Must be a valid [LANSA object name](#).
- Restricted to a maximum of 10 characters.
- Logical file names must be unique. No other physical or logical file can exist in the same library with the same name.
- Note that no library name is required. LANSAs will always create the logical file into the same library as the associated physical file. This is the library that was specified when the file definition was first created.

Platform Considerations

- IBM i: If using native RPG programming language, a maximum of 8 character names should be used.

Tips & Techniques

- Many IT departments adopt a naming standard that requires that logical file names contain or relate to the associated physical file name. This is to prevent confusion amongst users when they are accessing the physical or logical files.
- For instance, the physical file (and LANSAs file definition) of a customer master file might be named CUSMST. The associated logical views might be called:

CUSMSTV1 Customers ordered by customer name.

CUSMSTV2 Customers ordered by postcode.

CUSMSTV3 Customers ordered by state.

[↑ 3.4.1 Logical View Definition](#)

Logical View Description

Mandatory.

Specify the description to be associated with the logical view. This description is used within LANSAs and within the operating system to aid system users in identifying the file.

Rules

- A file description must be entered for each language defined for the partition.
- Maximum length is 40 characters.

Tips & Techniques

- Wherever possible include information in the description that specifies what the logical file/view should be used for. For example, "Customer master ordered by post code" or "Orders by customer number, order number".

[↑ 3.4.1 Logical View Definition](#)

Unique Key

Mandatory.

Specify whether or not the key fields nominated are to form a unique key to the file.

Default=NO.

Rules

- NO (unchecked or not selected), indicates that the key fields nominated do not form a unique key to the file. This allows multiple records with the same key to exist in the file.
- YES (checked or selected), which indicates that the key fields are to form a unique key to the file. This means that one (and only one) record can exist in the file for any given key value.

Tips & Techniques

- When defining a new logical file over a physical file that already contains records and using the YES option make sure that there are no duplicate records (i.e.: key values) already in the file. If duplicate records do exist, then the "make operational" job will fail as this logical view cannot be loaded because of the duplicate records. The job log will indicate the cause of this problem. To correct, remove or change the duplicate records.
- LANSAs automatically handles duplicate key errors and there is no need for user logic to handle or check for them.

[↑ 3.4.1 Logical View Definition](#)

Access Path

Mandatory.

Specify the method to be used to maintain the IBM i database access path associated with this logical view/file.

When a logical view is created to arrange the information in a file into a specific order, the IBM i database creates an access path.

The access path is essentially the logical file "key fields" arranged in a special structure that allows:

- Extremely fast access to any individual record in the file using just the key(s) of the logical file.
- The processing of the records in order of the logical file key(s) without the need to first sort the file.

An access path exists for every logical view/file created. Every time a record in the file is added, deleted or updated the operating system must update all the access paths to this file. To do this it must use up part of the computer's time and available processing cycles. This of course degrades the performance of the system. The more access paths that are being maintained, the slower the system will run.

Default=Immediate.

Rules

Allowable values are:

IMMED The access path should be maintained immediately (i.e. whenever a record is added, updated or deleted from the file). This option is the most common. If the logical file is to be used in an interactive environment then IMMED should always be used.

DELAY The access path should only be maintained when the logical file is used. This type of access path lies dormant and is not maintained by the operating system until such time as someone needs to use the logical file. Thus it places no burden on the operating system until it is actually required. Typically, logical views that use this option are only used very infrequently to "sort" a file into a specific order.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

Tips & Techniques

- If the logical file is to be used in an interactive environment, then this option should be selected.

[↑ 3.4.1 Logical View Definition](#)

Dynamic Select

Mandatory.

Specify whether or not any select/omit tests specified on the lower portion of the screen are to be done at execution time.

Default=NO

Rules

- NO (unchecked/not selected), indicates that the dynamic select feature should not be used. In this case the access path associated with the logical file will contain only records that match the select/omit criteria specified.
- YES (checked/selected) , indicates that the dynamic select feature should be used. In this case the access path associated with the logical file will contain all records in the file. The select/omit testing should be done when the program reads the records from the file.

Tips & Techniques

- If you specify YES (checked/selected), then do not specify any select/omit tests or else the value will be automatically changed back to NO.

Platform Considerations

- IBM i: The dynamic select facility is a feature of the operating system. Using it can have significant overall performance benefits in some situations. For more information about this facility refer to the appropriate IBM supplied manual.

[↑ 3.4.1 Logical View Definition](#)

Alt. Seq.

Optional.

Specifies the name of an Alternate Collating Table to be used when sequencing the records for retrieval on a Keyed file.

Warning

- At File Creation time, you must have operational rights to the Alternate Collating Table. Alternate Collating sequences are not valid for key fields with a data type of packed decimal as it causes zoned key fields to default to unsigned sequence.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

Tips & Techniques

- The Alternate Collating Table name, if specified, is not validated to determine if it actually exists.
- When attempting to use any file that has been created with an Alternate Collating Table, the library list is used to locate the Table file.

[↑ 3.4.1 Logical View Definition](#)

Record Format

Mandatory.

Specify the record format name to be assigned to the logical file record.

When the file definition is first set up, this value is initialized to be the same as the logical file name.

Default=Logical view name.

Rules

- The name specified must be no more than 10 characters long.
- Name specified must conform to IBM i record format naming conventions.
- First character must be A to Z. Do not use # as first character. Subsequent characters should be A to Z, 0 to 9. Characters \$ and # are allowed but not recommended.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

[↑ 3.4.1 Logical View Definition](#)

CRTLF/CHGLF Parameters

Specify additional command parameters that are to be used by LANSAs when creating (CRTLF command) or changing (CHGLF command) the logical file.

Rules

- Parameters specified are checked for validity.

Tips & Techniques

- When specifying parameters, input them exactly as would be done when entering them through the IBM i command entry facility by using "keyword" (rather than positional) specification of any parameters required.

[↑ 3.4.1 Logical View Definition](#)

3.4.2 Logical View Keys

[Key Field Name](#)

[Key Field Details](#)

[Numeric Ordering](#)

[Key Order](#)

[Key Position](#)

[↑ 3.4 Logical Views](#)

Key Field Name

Mandatory.

Specify the name of one of the field(s) in the logical view.

Rules

- At least one key must be specified for the view definition.
- Any fields named as a key must first be defined as a field in the file definition.

[↑ 3.4.2 Logical View Keys](#)

Key Field Details

Display Only.

The fields details of the specified [Key Field Name](#) will be displayed including:

- [Description](#)
- [Type](#)
- [Length](#)
- [Decimals](#)

You can review this information to confirm the field details.

[↑ 3.4.2 Logical View Keys](#)

Numeric Ordering

Mandatory.

Specify, for numeric key fields only, additional details about how the key is to be ordered.

Default=Unsigned.

Rules

Allowable values are:

S (Signed) indicates the numeric fields should be stored taking into account their signs (+ or -).

U (Unsigned) indicates the numeric field should be stored without taking into account their signs. The numeric field is to be treated just like a character field.

A (Absolute value) indicates that the numeric field should be stored by its absolute value. Note that this is not the same (and does not always produce the same order) as the U option.

Tips & Techniques

- Character fields are always U (unsigned) fields. If you specify something other than U for a character field, it will be automatically changed to U.

[↑ 3.4.2 Logical View Keys](#)

Key Order

Mandatory.

Specify whether the associated key is to be stored in ascending or descending sequence (i.e. from lowest to highest or highest to lowest order).

Default=Ascending.

Rules

Allowable values are:

A (Ascending) store in lowest to highest order

D (Descending) store in highest to lowest order.

This is an optional field. If not specified, A is assumed.

[↑ 3.4.2 Logical View Keys](#)

Key Position

Mandatory.

Specify the order of the keys defined. Key fields are specified from top to bottom in order of their importance in the key hierarchy (that is, major to minor). Major keys come first.

Default=Next sequential number.

Rules

- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a key field/value is added. Sequence number is updated when the order of the key is updated in the list of keys.

[↑ 3.4.2 Logical View Keys](#)

3.4.3 Select/Omit Criterion

[Select/Omit](#)

[Select/Omit And/Or](#)

[Select/Omit Type](#)

[Select/Omit Field Name](#)

[S/O Operator](#)

[S/O Value](#)

[S/O Range](#)

Also See

[3.4.1 Logical View Definition](#)

[3.4.2 Logical View Keys](#)

[Select Omit Concepts](#) in the *Developer Guide*.

[↑ 3.4 Logical Views](#)

Select/Omit

Mandatory.

Specify whether the condition specified in the operation(s) field is to be used to select or omit records when found to be true.

Rules

Allowable values are:

SELECT Use operation to select records

OMIT Use operation to omit records

blank Form an "and" with previous SELECT/OMIT operation

Tips & Techniques

- Using SELECT or OMIT implies an OR relationship with the preceding select/omit statement. When adding more than one select/omit criteria, the use of a blank value indicates that the criteria is being in an AND relationship with previous criteria. Refer to [Select/Omit And/Or](#).

Also See

[Select/Omit Type](#)

[↑ 3.4.3 Select/Omit Criterion](#)

Select/Omit And/Or

This is an output field only.

It indicates whether the previous select/omit statement is ANDed or ORed with this statement.

When using select/omit statements specify SELECT,OMIT or blanks in the SELECT/OMIT column. Using SELECT or OMIT implies an OR relationship with the preceding select/omit statement.

Leaving the entry in the SELECT/OMIT column blank implies an AND relationship with the preceding select/omit statement.

To confirm the proper AND/OR condition has been specified, refer to the completed description of the condition. Refer to [Logical Views Tab](#) in the *User Guide*.

Also See

[Select/Omit](#)

[↑ 3.4.3 Select/Omit Criterion](#)

Select/Omit Type

Mandatory.

Specify the operation that is to be performed against the field nominated in the "field" field.

Default=Range.

Rules

Allowable values are :

- **RANGE(<low value> <high value>)** which indicates that the nominated field should be tested against the range of values <low value> to <high value>. The <low value> and <high value> specified can be a character literal in quotes (e.g. 'BALMAIN'), a numeric literal (e.g. 1.54) or the name of another field in the file definition. Refer to [S/O Range](#) and [S/O Range To](#).
- **VALUES(<value1> <value2> <value100>)** which indicates that the nominated field should be compared with the list of values specified. Up to 100 values can be specified in the list of values. The value fields specified can be a character literal in quotes (e.g. 'BALMAIN') or a numeric literal. Refer to [S/O Value](#).
- **COMP(<operator> <value>)** which indicates that the nominated field should be compared using <operator> to the <value>. Refer to [S/O Operator](#) and [S/O Value](#).

Allowable values for <operator> are:

EQ Equal to

NE Not equal to

LT Less than

NL Not less than

GT Greater than

NG Not greater than

LE Less than or equal to

GE Greater than or equal to

The value specified for <value> can be a character literal in quotes (e.g. 'BALMAIN'), a numeric literal (e.g. 1.54) or the name of another field in the

file definition.

- **ALL** which is only ever used as the last select/omit statement associated with a logical view/file. It indicates what is to happen after all other select/omit statements have been processed. If used with SELECT then all records not meeting the previous select/omit statements will be selected. If used with OMIT then all records not meeting the previous select /omit statements will be omitted.

If the ALL condition is NOT specified as the last statement in a set of select/omit statements then a default value is assumed. The default value is the reverse of the last select/ omit statement specified. If the last select/omit statement is SELECT then a default of OMIT ALL is assumed as the last statement. Likewise if the last select/omit statement is OMIT then a default value of SELECT ALL is assumed as the last select/omit statement.

[↑ 3.4.3 Select/Omit Criterion](#)

Select/Omit Field Name

Mandatory.

Specify the name of the field that is to be used in conjunction with the operations(s) to evaluate the select/omit expression.

This field must be specified if the [Select/Omit Type](#) is RANGE, VALUES or COMP.

Rules

- Field named must be defined in the file definition.
- For fields of type Nchar or Nvarchar, and fields with SUNI attribute, the only valid comparison is EQ or NE *SQLNULL. Other Select/Omit conditions are not supported.

[↑ 3.4.3 Select/Omit Criterion](#)

S/O Operator

Mandatory.

Specify the operator to be used with the nominated [Select/Omit Field Name](#) and [S/O Value](#).

This field must be specified if the [Select/Omit Type](#) is **Compare**.

Rules

- Allowable values are:

EQ Equal to

NE Not equal to

LT Less than

NL Not less than

GT Greater than

NG Not greater than

LE Less than or equal to

GE Greater than or equal to

[↑ 3.4.3 Select/Omit Criterion](#)

S/O Value

Mandatory.

Specify the value that the select/omit field should be compared to.

This field must be specified if the [Select/Omit Type](#) is **Values** or **Compare**.

Rules

Allowable values are:

- Character literal in quotes (e.g. 'BALMAIN').
- Numeric literal (e.g. 1.54).
- DateTime literal in quotes. (e.g. '2005-05-01 10:00:00')
DateTime literals need to be entered as per the SUTC attribute of the field.
If the SUTC attribute is enabled, enter the value as UTC, otherwise enter the value in local time.
- Name of another field in the file definition.

[↑ 3.4.3 Select/Omit Criterion](#)

S/O Range

These fields must be specified if the [Select/Omit Type](#) is **Range**.

Range From

Mandatory.

Specify the lower value that the select/omit field should be compared to.

Rules

Allowable values are:

- Character literal in quotes (e.g. 'BALMAIN').
- Numeric literal (e.g. 1.54).
- DateTime literal in quotes. (e.g. '2005-05-01 10:00:00')
DateTime literals need to be entered as per the SUTC attribute of the field.
If the SUTC attribute is enabled, enter the value as UTC, otherwise enter the value in local time.
- Name of another field in the file definition.

S/O Range To

Mandatory.

Rules

Allowable values are:

- Character literal in quotes (e.g. 'BALMAIN').
- Numeric literal (e.g. 1.54).
- DateTime literal in quotes. (e.g. '2005-05-01 10:00:00')
DateTime literals need to be entered as per the SUTC attribute of the field.
If the SUTC attribute is enabled, enter the value as UTC., otherwise enter the value in local time.
- Name of another field in the file definition.

[↑ 3.4.3 Select/Omit Criterion](#)

3.5 Access Routes

LANSA uses access routes to describe relationships between files in a database. They provide information about the database map or schema that can be used by the LANSAs development environment and other products. Access routes also support Predetermined Join Fields which are a special type of virtual field.

[3.5.1 Access Route Definitions](#)

[3.5.2 Predetermined Join Field Definitions](#)

Also See

[Access Route Tab](#) in the *User Guide*.

[Access Route Development](#) in the *Developer Guide*.

[Predetermined Join Field Development](#) in the *Developer Guide*.

↑ [3. Files](#)

3.5.1 Access Route Definitions

Access Route Name	Association Rule	Derivation
Access Route Description	Documentation Only	Key Fields/Values
Accessed File	Maximum Records	Key Field Details
Association Type	Default Action	Target Field Details
		Key Position

Also See

[Access Route Tab](#) in the *User Guide*.

[Access Route Development](#) in the *Developer Guide*.

[↑ 3.5 Access Routes](#)

Access Route Name

Mandatory.

Specify the name that is to be assigned to the access route.

This name is used for identification purposes only.

Rules

- The name specified must be unique within the current file definition.
- Maximum length is 10 characters.

Tips & Techniques

- It is suggested that a naming convention be developed for access route names. For example, first 3 letters may indicate current file definition, next 3 indicate the name of the file being accessed, etc.

[↑ 3.5.1 Access Route Definitions](#)

Access Route Description

Mandatory.

Specify the description of this access route to assist users in identifying what information can be obtained from the access route.

For example:

Current File Definition	File Accessed Via Access Route	Access Route Description
Order header	Order lines	Order lines associated with order
Order header	Cust master	Full details of order customer
Cust master	Order header	Orders associated with this customer
Order lines	Order header	Order associated with this order line

Rules

- Maximum length is 40 characters.

Tips & Techniques

- Access route information can be used by reporting tools such as LANSA Client. End users will benefit from proper access route descriptions.

[↑ 3.5.1 Access Route Definitions](#)

Accessed File

Mandatory.

Specify the name of the physical or logical file that is to be accessed via this access route.

Rules

- The physical or logical file specified must exist in the repository.

Also See

[Key Fields/Values](#)

[↑ 3.5.1 Access Route Definitions](#)

Association Type

In conjunction with Association Rule, Association Type identifies the nature of the relationship between the current file and the target file as specified in the Accessed File property. They allow a more precise meaning to be ascribed to an Access route than was previously possible with the Maximum Records and Default Action features, and are used by the Database Diagram Viewer to better show relationships.

There are several possible values:

- Join
- Part Of, Parent
- Child
- Whole Part
- Derive from Maximum Records
- Default Action.

Changes to Association Type and Rule may cause the Maximum Records and Default Action to automatically update.

Association Rule

In conjunction with Association Type, Association Rule identifies the nature of the relationship between the current file and the target file as specified in the Accessed File property. They allow a more precise meaning to be ascribed to an Access route than was previously possible with the Maximum Records and Default Action features, and are used by the Database Diagram Viewer to better show relationships.

There are three possible values:

- Optional
- Mandatory
- Derive from Default Action.

Changes to Association Type and Rule may cause the Maximum Records and Default Action to automatically update.

Documentation Only

As the name suggests, an Access Route flagged as Documentation Only is simply defined for the purposes of documenting a relationship between two tables so that a relationship can be displayed in the Database Diagram Viewer.

Documentation Only Access Routes are not actionable in anyway and will not be compiled in to the OAM. They will not appear as relationships in LANSAClient and do not support the definition of Predetermined Join Fields

Maximum Records

Mandatory.

Specify the maximum number of records that are expected to be found in the [Accessed File](#) that have a key matching the [Key Fields/Values](#) specified.

If the value is **One** then a "1:1" relationship between the files is established. If the value is **More than one**, then a "1:many" relationship is established. The relationship between the files affects the method by which screen formats are designed. It determines the type of PJFs that can be created. (Refer to [Access Routes and PJFs](#) in the *Developer Guide*.)

Rules

Allowable values are:

- One
- More than one

Platform Considerations

- IBM i: Range of allowed values is 1 to 9999.

Also See

[Keep Last](#)

[Key Fields/Values](#)

[Predetermined Join Field Development](#) in the *Developer Guide*.

[↑ 3.5.1 Access Route Definitions](#)

Keep Last

Specify the number of retrieved Predetermined Join Field values to be kept in memory.

This value applies to Predetermined Join Fields defined on the access route when the relationship is one to one. (Refer to [Maximum Records](#).) Each value retrieved from the accessed file is stored in memory up to the keep last value.

Rules

- Range of values is 0 to 999.
- A value can only be entered if the access route relationship is one to many, i.e. the [Maximum Records](#) is set to **One**.

Tips & Techniques

- This is a very useful feature when using Predetermined Join Fields to retrieve values from small frequently used code fields to reduce I/Os.
- By storing values in memory, file accesses can be reduced and performance is improved. If more than the keep last value are retrieved, then a file access is simply performed and the current values are overwritten starting from the first value retrieved.

Also See

[Predetermined Join Field Development](#) in the *Developer Guide*.

[PJF Type](#)

[↑ 3.5.1 Access Route Definitions](#)

Default Action

Mandatory.

Specify the actions to take if no records can be found in the [Accessed File](#) that have a key matching the [Key Fields/Values](#) specified.

Default= Ignore and continue processing (IGNORE).

Rules

Allowable values are:

- **Abort and issue an error message (ABORT):** The function attempting to access the file specified should abort (fail) with an error message indicating the cause of the problem. This option can be used to continually verify database integrity. For instance, an access route from an order lines file to an order header file should always find a record. An order line without an associated order header probably indicates database corruption.
- **Ignore and continue processing (IGNORE):** The function attempting to access the file specified should ignore the no records situation and continue to process. This option may be valid in the reverse access path of the case above. It is perfectly valid for an order header record to have no associated order lines.
- **Create a dummy record with empty fields (DUMMY):** The function attempting to access the file specified should create a "dummy" record when no "real" record(s) can be found. The dummy record created will have blanks in all alphanumeric fields and zero (0) in all numeric fields. Only one dummy record will be created.
- **Create a dummy record and mark fields as not available (N/AVAIL):** The function attempting to access the file specified should create a "dummy" record when no "real" record(s) can be found. The dummy record created will have zero (in all numeric fields, blanks in all alphanumeric field less than 3 characters long, and as much of the string as will fit in all other alphanumeric fields. Only one "dummy" record will be created.

Tips & Techniques

- The N/AVAIL option is useful in situations where the no record situation arises occasionally. For instance an access route from an archived invoices file to a customer mast file may use this option. When the name of a customer associated with an archived invoice cannot be found (presumably because it has been deleted) then it will be displayed as N/AVAIL, rather than causing an

error.

[↑ 3.5.1 Access Route Definitions](#)

Derivation

Mandatory.

Specify whether you want predetermined join field values to be derived before or after the virtual field values are derived.

Default=After virtual fields.

Rules

Allowable values are:

- Before virtual fields
- After virtual fields

Tips & Techniques

- This option allows a Predetermined Join Field to be used in deriving a virtual field or a virtual field to be used as a key to a file accessed for Predetermined Join Fields.

Also See

[3.3 Virtual Fields in File](#)

[Predetermined Join Field Development](#) in the *Developer Guide*.

[↑ 3.5.1 Access Route Definitions](#)

Key Fields/Values

Mandatory.

Specify the fields or values that should be used to form the key that will be used to access the record(s) in the [Accessed File](#).

Up to 20 key fields or values can be specified. Key values are checked for type and length compatibility. The entire key list supplied is checked for compatibility with the actual key(s) of the Accessed File.

Rules

Allowable values include:

- A field that is defined in the current file definition.
- An alphanumeric literal (in quotes) such as 'NSW', 'BALMAIN'
- A numeric literal such as 1, 14.23, -1.141217.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.

Tips & Techniques

- In cases where a "1:many" relationship is being defined, the key list specified would almost always be partial key to the file. Refer to [Maximum Records](#).
- In a 1:1 relationship, it may be valid to only supply a partial key. However, if the warning message is issued, check what has just been defined.
- It should be noted that the access route facility is provided as an aid to user traversal of database structures. LANSAR DML does not use access routes in any way nor does it restrict database access to pre-defined access routes.

Also See

[Key Position](#)

[↑ 3.5.1 Access Route Definitions](#)

Key Field Details

Display Only.

The fields details of the specified [Key Fields/Values](#) will be displayed including:

- [Description](#)
- [Type](#)
- [Length](#)
- [Decimals](#)

You can review this information to confirm compatibility with the [Target Field Details](#) of the accessed file.

Also See

[Key Fields/Values](#)

[↑ 3.5.1 Access Route Definitions](#)

Target Field Details

Display Only.

The fields details of the target field in the accessed file will be displayed including:

- [Description](#)
- [Type](#)
- [Length](#)
- [Decimals](#)

You can review this information to confirm compatibility with the [Key Fields/Values](#) of the accessed file.

Also See

[Key Fields/Values](#)

[↑ 3.5.1 Access Route Definitions](#)

Key Position

Mandatory.

Specify the sequence of the entered key field/value.

Rules

- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a key field/value is added. Sequence number is updated when the order of the key is updated in the list of keys.

Also See

[Key Fields/Values](#)

[↑ 3.5.1 Access Route Definitions](#)

3.5.2 Predetermined Join Field Definitions

[PJF Field](#)

[PJF Field Details](#)

[PJF Type](#)

[PJF Source Field](#)

Also See

[Predetermined Join Field Development](#) in the *Developer Guide*.

[↑ 3.5 Access Routes](#)

PJF Field

Mandatory.

Specify the name of the predetermined join field.

No check is made on the Predetermined Join Field's suitability to hold the required information.

Rules

- It must exist in the repository.
- It must not already exist as a Predetermined Join, Virtual or Real field on the file.
- It must be of the type allowed by [PJF Type](#). Field type must be numeric if the [PJF Type](#) is Total, Maximum, Minimum, Average or Count.
- It must be a suitable type to match the [PJF Source Field](#).

Tips & Techniques

- The Predetermined Join Field may have the same name as the [PJF Source Field](#) as long as it complies with the defined rules.
- It is your responsibility to check that Predetermined Join Field is suitable to hold the required information. For example, ensure fields that hold counts or totals are large enough.

[↑ 3.5.2 Predetermined Join Field Definitions](#)

PJF Field Details

Display Only.

The fields details of the specified [PJF Field](#) or [PJF Source Field](#) will be displayed including:

- [Description](#)
- [Type](#)
- [Length](#)
- [Decimals](#)

You can review this information to confirm the field details.

[↑ 3.5.2 Predetermined Join Field Definitions](#)

PJF Type

Mandatory.

Specify the type of predetermined joined field to be assigned to the [PJF Field](#).

The PJF type is checked against the [Maximum Records](#) specified for the access route.

Rules

- Fields with value of *SQLNULL are ignored.

Allowable values are:

- Lookup** Specifies that the predetermined join field will hold the value retrieved from the selected [PJF Source Field](#) in the accessed file. This option is only available when there is a 1:1 relationship between the files. Refer to [Maximum Records](#) and [Keep Last](#). The predetermined join field must be of the same type as the selected field.
- Total** Specifies that the predetermined join field is to hold the total of the selected [PJF Source Field](#) retrieved using the access route key. This option is only available when there is a 1:many relationship between the files and the selected field is numeric. Refer to [Maximum Records](#).
- Maximum** Specifies that the predetermined join field is to hold the highest value of the [PJF Source Field](#) selected field in records retrieved using the access route key. This option is only available when there is a 1:many relationship between the files and the selected field is numeric. Refer to [Maximum Records](#).
- Minimum** Specifies that the predetermined join field is to hold the lowest value of the selected [PJF Source Field](#) in records retrieved using the access route key. This option is only available when there is a 1:many relationship between the files and the selected field is numeric. Refer to [Maximum Records](#).
- Average** Specifies that the predetermined join field is to hold the average

value of the selected [PJF Source Field](#) in records retrieved using the access route key.

This option is only available when there is a 1:many relationship between the files and the selected field is numeric. Refer to [Maximum Records](#).

Count Specifies that the predetermined join field is to hold the count of the number of fields retrieved using the access route key.

This option is only available when there is a 1:many relationship between the files. Refer to [Maximum Records](#).

Also See

[PJF Source Field](#)

[↑ 3.5.2 Predetermined Join Field Definitions](#)

PJF Source Field

Mandatory.

Specify the source field in the [Accessed File](#) that is to be used with the operation specified by the [PJF Type](#).

No check is made on the Predetermined Join Field's suitability to hold the required information.

Rules

- Field must exist in the nominated [Accessed File](#) of the access route.
- Field type must be numeric if the [PJF Type](#) is Total, Maximum, Minimum, Average.

Tips & Techniques

- It is your responsibility to check that the [PJF Field](#) is suitable to hold the required information.

Also See

[PJF Field](#)

[3.5.1 Access Route Definitions](#)

[↑ 3.5.2 Predetermined Join Field Definitions](#)

3.6 File Attributes

When a file is defined in the Repository, some operating system specific file "attributes" that influence how the file is to be set up and used can be specified.

Some of the attributes influence:

- How big the file is allowed to be
- When the file is to be recovered after a system failure
- Whether space should be allocated for the file when it is created
- Whether or not the file is to be under commitment control

Refer to:

3.6.1 File Library	3.6.9 Strip Debug	3.6.17 Convert Special Characters in Field Names
3.6.2 Record Format Name	3.6.10 Suppress IOM0034 Message	3.6.18 Commitment Control
3.6.3 I/O Module Library	3.6.11 Ignore Decimal Errors	3.6.19 Auto Commit
3.6.4 File Uses SQL On IBM i	3.6.12 IOM Required	3.6.20 CRTPF and CHGPF Parameter
3.6.5 Alternate Collating Table (ALTSEQ)	3.6.13 Create Batch Control	3.6.21 Readonly Access
3.6.6 Enable Files for RDMLX	3.6.14 IBM i High Speed Table	3.6.22 Database File Trigger
3.6.7 Share	3.6.15 Auto RRN Creation	3.6.23 File Description
3.6.8 Secure	3.6.16 Create RRNO Column	

[↑ 3. Files](#)

[File Attributes Tab](#) in the *User Guide*.

3.6.1 File Library

Mandatory.

Specify the library in which the new file is to reside.

This field is pre-filled with the partition default file library name.

Rules

- Library name must conform to object naming standards for the operating system.
- When the library name is combined with the file name it must form a unique name for the file.

Warnings

- The library specified must exist and you must be authorized to use it. LANSAP does not check whether the library specified exists or not.

Platform Considerations

- IBM i: Avoid creating files that reside in your IBM i "current" library.

[↑ 3.6 File Attributes](#)

3.6.2 Record Format Name

Mandatory.

Specify the record format name to be assigned to the physical file record. When the file definition is first set up, this value is initialized to be the same as the file name.

Rules

- The name specified must be no more than 10 characters long.
- Name specified must conform to IBM i record format naming conventions.
- First character must be A to Z. Do not use # as first character. Subsequent characters should be A to Z, 0 to 9. Characters \$ and # are allowed but not recommended.

Tips & Techniques

- It is recommended that no more than 8 characters are used.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

[↑ 3.6 File Attributes](#)

3.6.3 I/O Module Library

Mandatory.

Specify in which library the file's I/O module will reside.

Rules

Allowable values are:

- Same library as the file (F)
- Partition module library (M)

[↑ 3.6 File Attributes](#)

3.6.4 File Uses SQL On IBM i

Read only.

For LANSAs Files, Yes indicates that the IBM i file was created with SQL the last time it was built (rather than DDS and CRTPF). This value is only set on Checkout from IBM i (or when a LANSAs Import is done).

LANSAs files are built with SQL for a number of reasons, for example, if they include a BLOB or CLOB field. You can force all files to be built with LANSAs SQL. For details, see the option *Always Build Using SQL (IBM i)* in [System Definitions](#) in the *Administrators Guide*.

For IBM i Other Files, Yes indicates that SQL must be used to access the file at execution time. This value is set during Load Other File. Refer to [Load Other File](#) for further information.

Also see

Always build using SQL in [Field and File Defaults](#) in the *LANSAs for i User Guide*.

[↑ 3.6 File Attributes](#)

3.6.5 Alternate Collating Table (ALTSEQ)

Specify the name of an Alternate Collating Table to be used when sequencing the records for retrieval on a Keyed file.

Default = blank (none).

Rules

- Table name must conform to object naming standards for the operating system.

Warnings

- At File Creation time, you must have operational rights to the Alternate Collating Table. Alternate Collating sequences are not valid for key fields with a data type of packed decimal. Causes zoned key fields to default to unsigned sequence.

Tips & Techniques

- The Alternate Collating Table name if specified is not validated that it actually exists.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.
- IBM i: When attempting to use any file that has been created with an Alternate Collating Table, the library list is used to locate the Table file.

[↑ 3.6 File Attributes](#)

3.6.6 Enable Files for RDMLX

To enable a file for RDMLX, open the file in the Visual LANSA Editor and choose the *File Attributes* tab. Select the *Enabled for RDMLX* option.

This option is only available in an RDMLX Enabled Partition.

The default value for this option is controlled in the [RDMLX Partition Settings](#) described in the *Administrator Guide*.

Tips & Techniques

- If you change this file option, the file definition must meet all requirements before it can be saved.
- Performance characteristics of file operations may change and should be properly evaluated once the conversion to RDMLX has been made.
- It is recommended that you review the [RDML and RDMLX Partition Concepts](#) information in the *Administrator Guide*.

Implications:

- RDMLX files **cannot** be used by LANSA RDML Object Types. RDML Files **can** be used by RDMLX objects.
- You cannot import RDMLX objects into RDML partitions

Warning

- If you change an RDMLX file back to an RDML file but you must remove all RDMLX features before you save the file.
- All editing must be performed using Visual LANSA. RDMLX Files cannot be edited from LANSA for i.

[↑ 3.6 File Attributes](#)

3.6.7 Share

Specify whether or not this file (and any of its associated logical views) should be opened with the option to share an open data path.

Note that this value only affects the way that the I/O module (and hence RDML applications) open the file at run time. To actually get the physical file and logical files defined to the operating system as SHARE(*YES) or SHARE(*NO) for the benefit of external application programs, refer to the [3.6.20 CRTPF and CHGPF Parameter](#).

Default = NO (unchecked/not selected).

Warnings

- This option relates to the opening of the file's open data path when used in I/O modules and *DBOPTIMIZE programs.
- This option does not relate to the creation or changing of the database file attributes.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.
- IBM i: Despite the fact that IBM recommends data path sharing as one of the fundamental design considerations for efficient applications, the default for LANSA created files is NO (UNCHECKED or unselected). That is, do not share open data paths. The reason for this is that the I/O module concept implicitly emulates an open data path, because one active I/O module is shared by all RDML functions (within a job) that are accessing a file.
- IBM i: Generally use option NO (UNCHECKED/not selected). That is do not share an open data path, except in the special situation where the file is only being used as a 'joined record format' for data that is dynamically created by the IBM OPNQRYF (open query file) command.

[↑ 3.6 File Attributes](#)

3.6.8 Secure

Specify whether or not this file (and any of its associated logical views) should be opened with the option to be secured from file override commands already issued by higher level program(s).

Default = YES (checked/selected).

Warnings

- The default for this field is YES (checked/selected). Do not change this value unless you understand what it does and how it is used. Refer to the appropriate IBM supplied manual(s) for more information about the SECURE parameter on file override commands and the effects of using it.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

[↑ 3.6 File Attributes](#)

3.6.9 Strip Debug

Specify whether or not the I/O module associated with the file definition should have its associated debug symbolic information removed.

Default = YES (checked/selected).

Tips & Techniques

- This information is only required when you are using the IBM supplied debugging aids with the compiled RPG I/O module.
- Since these situations are relatively rare, the default for this field is YES (debugging information should be stripped). By using this option, the size of the compiled I/O module will typically be reduced by 40 to 60%. This size reduction has no bearing on execution speed, just on the size of the compiled object.

Platform Considerations

- IBM i: To enable I/O modules to be debugged, or in environments where they will be ported from CISC IBM i to RISC IBM i, use option NO (debug information should not be stripped).

[↑ 3.6 File Attributes](#)

3.6.10 Suppress IOM0034 Message

Specify whether or not to suppress the IOM0034 Message.

When an I/O module is used to access a file that has never been directly defined to LANSAs, it issues message IOM0034 indicating that no LANSAs level security information exists for the file.

To suppress this message specify YES (checked/selected). To allow this message to be issued, specify NO (UNCHECKED or not selected). That is, message is not to be suppressed..

Default = NO (unchecked/not selected).

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

[↑ 3.6 File Attributes](#)

3.6.11 Ignore Decimal Errors

Specify whether or not the I/O module associated with the file definition should be compiled with the IGNDECERR (ignore decimal data error) option.

Default = NO (unchecked/not selected).

Warnings

- While the IGNDECERR option is available in LANSAs, the IBM warnings about using this option still apply. Should using this option cause problems your product vendor may assist you in correcting them, but obviously cannot accept any responsibility for the problem or its cause.
- Do not use this option unless absolutely necessary. Refer to the IBM i CRTRPGPGM command for more details of the IBM warnings and disclaimer before attempting to use this option.

Tips & Techniques

- The default is NO (not checked/not selected). The use of YES (checked/selected) is not recommended.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

[↑ 3.6 File Attributes](#)

3.6.12 IOM Required

Specify whether or not an I/O module is required for this file and its associated logical views.

To effectively use NO (not checked/not selected), you must use the FUNCTION RDML command with the OPTIONS(*DBOPTIMISE) keyword in every RDML function that attempts to access this file or any of its associated logical views. If this command is omitted from the RDML function, it will still compile correctly, but at execution time, it will fail because it places a call to the non-existent I/O module.

Default = YES (checked/selected).

Warnings

- Note that if this value is changed from YES(checked/selected) to NO (unchecked/not selected), and the resultant change made operational, any existing I/O module will be automatically deleted. After this has been done, all existing RDML functions that access this file without using *DBOPTIMISE will fail when used. This is because they are attempting to resolve to a now non-existent I/O module. In such cases, add the *DBOPTIMISE option to the RDML functions involved and recompile them.

Tips & Techniques

- It is strongly recommended that you do not use NO (not checked/not selected) until you have had some experience with the LANSAs product and are familiar with the concept and workings of I/O modules.
- Refer to the [FUNCTION](#) command and [Using *DBOPTIMIZE / *DBOPTIMIZE_Batch](#) in the *LANSAs for i User Guide* for more details of how to set up and use a system without using I/O modules.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

[↑ 3.6 File Attributes](#)

3.6.13 Create Batch Control

Specify whether or not other database files that perform batch control totaling against this file should be allowed to automatically create missing "batch" records in this file.

NO, indicates that when a batch control record cannot be found in this file, the other file operation should fail with error message IOM0032.

YES, indicates that when a batch control record cannot be found in this file, the other file's I/O operation is allowed to automatically create one. The other file's I/O operation is modified to create the missing batch control record in this file like this:

- All fields in this file's record format are set to blanks (alphanumeric fields) or zero (numeric fields).
- The field(s) or value(s) from the other file that were used as key(s) to access the batch control record in this file are mapped into their corresponding field(s) in this file. This mapping is by key matching, not by name.
- The new record thus assembled is inserted into this file.

Default = NO (unchecked/not selected).

Warnings

- Before attempting to use this option ensure that you understand the ramifications of using it and then thoroughly test all resulting applications.
- When changing this option, it is necessary to make the amendment to this file operational. Additionally, all I/O modules for other files that perform batch control totaling against this file must also be (re)compiled.

[↑ 3.6 File Attributes](#)

3.6.14 IBM i High Speed Table

Specify whether this file definition and associated logical views should be mirrored into a high speed IBM i User Index to allow more rapid access in "read only" situations.

The following points provide basic information about the IBM i high speed table facility. They should all be read and understood before this facility is used in any way:

- A high speed table is not a "thing" in itself. A high speed table is a normal LANSAs file definition that has its "high speed" flag set to YES.
- A LANSAs file definition flagged as high speed table is actually implemented as a normal database file. All functions which insert, update or delete data in the table actually access the normal database file.

This normal database file actually contains the data, so there is no difference in the risk of data loss between a normal file definition and a high speed table. This also means that you can, at any time, set the high speed table flag back to NO and revert to a normal database file again without any loss of definition or data.

The difference between a normal file and a high speed table is that a high speed table uses an extra object (over and above the normal database file). This object is called an IBM i "User Index" and it contains a duplication or "mirror" of the data in the associated database file and its logical views. Functions that only read the file actually access the "mirror" data in the user index. Such a method of access has some strong advantages:

- It is very fast.
- There is no open or close overhead.
- There is very little of the normal space (PAG) overhead associated with having a normal database file open.

In a traditional commercial application that had, say, 40 database files open, there is a significant overhead in space and time to open and keep open the 40 files.

However, if 20 of these files were implemented as high speed tables, then the space and time overheads would be reduced by a factor of approximately 50%. This would probably significantly enhance the performance of this application. The high speed table index is an IBM i "platform specific" option. On other platforms the "high speed table" flag may be ignored and the file would be implemented just like any other file.

This may be a design consideration. Do not use the very high speed of IBM i User Indexes to "over-engineer" an application to the point that it will not be able to function on other platforms using normal database management facilities. Details of the IBM i User Index facility and its use can be obtained from the IBM i Information Center. It is possible for existing 3GL based application programs to also access the high speed tables.

Default = NO (unchecked/not selected).

Usage Rules

To be valid as a high speed table a file definition must conform to the following rules. Most of these things are checked during the "make operational" phase of creating/changing a file. If a rule is violated the make operational will fail with appropriate error message(s).

These rules apply to the basic physical file definition and all logic views defined over it:

- No form of alternate collating sequence is supported. The IBM i User Index facility only supports simple binary collation.
From "SC21-8226", the manual that elaborates upon access to an IBM i User (or Independent) Index:
"Each entry is inserted into the index at the appropriate location based on the binary value of the argument. No other collating sequence is supported."
- All key fields must be ascending, unsigned values.
- When a file with date, time or timestamp fields in its key list is mirrored in a high speed table, a LANSAs function with read-only access to the file will not use the I/O module. The date, time or timestamp field is treated as an alphanumeric field in the high speed table. Therefore values must be entered in full (for example, as 1999-01-02 not as 1999-1-2) when fetching a record. Also, if an invalid value is entered, the LANSAs function will not check if it is a valid date, time or timestamp, but just return a not-found status.
- The table can have no more than 99 fields.
- The maximum table entry record length depends on the system data area DC@OSVEROP. If option *HSTABEXTEND has been inserted, the maximum entry record length is 1988 bytes (this is an OS400 limit) and a maximum key length of 108 bytes (this is a LANSAs limit for storage and performance reasons). The key is included in the 1988 record length. If option *HSTABEXTEND is not in the system data area, the table entry record length cannot exceed 108 bytes. WARNING entry record lengths greater than 108

bytes cannot be saved to or restored from an OS400 release prior to V2R2M0. Note for packed fields their decimal length is counted, rather than their byte length. For more information refer to

[Allow extended files to be added to HST](#) in Compile and Edit Settings.

- The base physical file must have one or more primary key fields.
- The concepts of file members, run time library list changes and any form of file override or rename are NOT supported in any way in the high speed table execution environment. There is one high speed table index per LANSA partition. When an application is invoked that needs to access the index, it uses the single index associated with the current partition.
- No select omit logic can be specified.
- No batch control logic can be specified.
- No form of open, read or close trigger can be specified for any field in the file, either at the field or file level.
- No virtual fields or logic (code) can be defined.
- No read security will be actioned for the table. This means that a function cannot be stopped from reading the content of the high speed table. However, they can be stopped from modifying it in the normal manner (because they are actually modifying the normal database file, not the high speed index). This restriction exists to ensure maximum performance in read only applications. Applying read security would severely impact the performance of tables where only a few accesses are made. In fact the security checking time would be far longer than the actual time taken to access many table entries.
- Functions that modify (INSERT, UPDATE or DELETE) files that are tagged as high speed tables cannot use *DBOPTIMISE, *DBOPTIMISE_BATCH or any other option that infers these options. This restriction exists because the special logic required to "mirror" the real file data into the high speed index only exists in the associated I/O module. Thus all "table modifiers" must be forced to use the appropriate I/O module.
- Functions that only read from a high speed table may use *DBOPTIMISE or *DBOPTIMISE_BATCH in the normal manner.
- When the definition (i.e.: layout) of a high speed table is changed all functions and I/O module validation rules that read from the high speed table rather than the real file need to be recompiled. Again, this restriction exists to provide maximum performance.

By definition tables are largely static in design and content, so this should not be a problem. If it proves to be, remove the high speed table option from the definition of the file.

- No form of locking is supported in applications that only read from high speed tables. If you need record locking in a "read only" function, then your file is not a good candidate for the high speed table facility.
- The use of any of the following facilities with high speed tables is not checked, but they are not supported in any form within functions that require "read only" access to high speed tables:
 - The use of the OPEN command with the *OPNQRYP option.
 - The use of the *BLOCKnnn option in any form.
 - The use of SELECT_SQL in any form.
 - The use of WITH_RRN, RETURN_RRN or any form of relative record addressing.
 - The ISS_MSG parameter in any form.

In summary, the high speed table facility is designed for use with "plain vanilla" lookup and decode style files only. Files that are to be used in any other "fancy" way at all should not be implemented as high speed tables.

Tips & Techniques

Some common questions about high speed tables:

Q: What type of files are candidates to be high speed tables?

Broadly speaking, database files that have the following characteristics are good candidates for high speed table implementations:

- The data content is widely used for decode (e.g.: state code 'CA' is printed as description "CALIFORNIA") and validation (e.g.: is state code 'CA' valid?).
- The data content is relatively stable. (e.g.: How often is a new state acquired). Generally this means files that are not subject to continual and random change on a daily basis. A "product" file would be a good candidate if it only contained descriptive details because products are not created/changed often. However if it contained stock levels it would not be a good candidate because stock levels are continually changed.
- There are usually a small number of records in the file (say, for example, 5000 or less).
- There is usually only one application that "maintains" the file, and it is not used often (say, once per day or less often).

- The vast bulk of applications only "read" from the file for decode and validation purposes.

Q: Where is the high speed table data kept?

A LANSAs file definition flagged as a high speed table is set up just like any other file. The actual file data is stored and maintained in this normal file. However, the data is also mirrored into a "read only" high speed index to allow very fast access from "read only" applications.

The high speed index is actually an IBM i User Index (object type *USRIDX). It is automatically created in, and must always remain in, the module (or program) library of the current partition. You do not have to create this index, but you may choose to periodically delete and rebuild it. See the following points for an example of this. It is named DC@TBLIDX.

Q: Do I need to backup the high speed index?

Not really. Since each individual table has an associated data real file containing the "real" data, then you can actually re-create the high speed index for all tables in just a few minutes by using the built in function REBUILD_TABLE_INDEX. However, a synchronized backup of the index and all the associated database files containing the "real" data may simplify and speed up your restore procedures, should they need to be invoked.

Q: When is the high speed index accessed?

At various points the LANSAs code translators may generate code to access database files. When this is done and the file involved is a high speed table, then the high speed index will actually be accessed instead of the real file in the following situations :

- In RDML functions that only "read" from the file via CHECK_FOR, FILECHECK, FETCH or SELECT commands. When an RDML function is compiled it is checked for direct access to a high speed table. If all accesses to all the high speed tables used in the function are "read only" then the I/O will be directed to the high speed table rather than to the real database file.
- In inter file validation checks. I/O modules or *DBOPTIMISE generated code that needs to lookup a file entry as the result of a validation check will always look in the high speed index rather than the real file.

Q: Can I use *DBOPTIMISE/*DBOPTIMIZE with high speed tables?

Yes you can in all situations except where the function updates a high speed table. Functions that update a high speed table must do all their I/O to the table

via the associated I/O module. This ensures that real data file and the mirrored high speed index are updated together.

Q: When is the high speed index updated?

When the I/O module for a file that is flagged as a high speed table is created extra code is added to it to count the number of inserts, updates and deletes performed to the file.

When the file is being closed this count is examined, and if greater than 0, all existing entries for the file are erased from the high speed index, then the real file (and its views) are read from end to end to insert new entries into the high speed index.

This architecture has some impacts on the use of high speed tables:

- The file and the mirror index are not actually maintained simultaneously. When the file is being closed the existing mirrored index entries are erased and then recreated from the updated version of the file.
- The file and all its views are maintained as separate high speed index data. This means that a table with 4 views actually uses 5 times the index space of the source table. One for the table and one for each of the views.
- Contention may occur if multiple users attempt to update a file that has a high speed index mirror simultaneously. This problem is easily overcome by ensuring that applications which update high speed tables are restricted to single user access.

There are a variety of simple methods that may be used to restrict a function to single user access. Contact your product vendor if you require assistance in designing such an application.

Q: Can the "real" file and the index get out of synchronization?

From the previous points it can be seen that it is possible for a file and its mirrored high speed index to get out of synchronization. For example, a function may insert 3 new entries to a table and then fail. At this point the new entries are in the real file but they are not reflected in the high speed index.

Q: How can the lack of synchronization be corrected?

If a file and its high speed index get out of synchronization then they may be resynchronized by:

- Doing a "dummy" update to the file. The associated I/O module will then rebuild the index to reflect the updated file thus synchronizing the file and index again.

- Use the built in function REBUILD_FILE_INDEX to manually trigger the I/O module to rebuild the index of one or more files.

In fact, this sequence of commands will physically delete the entire IBM i user index area and then rebuild the indices of all high speed tables within the current partition. The first file rebuild will recreate the IBM i user index if it does not currently exist.:

```
EXEC_OS400 CMD('DLTUSRIDX DC@TBLIDX')
USE BUILTIN(REBUILD_FILE_INDEX) WITH_ARGS('*ALL')
```

Q: What happens when I change the layout of a file?

If you change the layout of a file and then "make the change operational" a resynchronization of the table and index will be automatically performed. This automatic synchronization is not performed if you then export the changed definition to another system.

Q: What happens if I import a high speed table to another system?

A high speed table is imported to another system just like a normal file. However, if the file data is imported, or the file layout is changed, the associated index is not automatically updated/reformatted. To do this you should trigger a "resynchronization" of the file and its index using any of the techniques previously described.

Note: A user index greater than 1 gigabyte or with an entry record length greater than 108 bytes cannot be saved to or restored from an OS/400 release prior to V2R2M0.

Warnings

- It is strongly recommended that, if option *HSTABEXTEND is added to system data area DC@OSVEROP, to make either the extended entry record length available or remove it to limit entry length. All files tagged as high speed tables, all read only functions that use these files and all other I/O modules and DBOPTIMIZED functions that use high speed tables for lookup validation rules, must be recompiled AFTER deleting the current user index. This index is DC@TBLIDX if adding *HSTABEXTEND, or DC@TBLIDY if removing *HSTABEXTEND.
- If this is not done, all functions that use a particular file and the I/O module must be recompiled at the same time or they will not be pointing to the same index. The situation will be further complicated by I/O modules and DBOPTIMIZED functions which use high speed table files for lookup

validation rules also pointing to the wrong index. It may not be obvious to the user that there is a problem as the database file and one index will be unsynchronized, but it will not cause program failure.

Platform Considerations

- IBM i: This file attribute applies to IBM i databases only.

[↑ 3.6 File Attributes](#)

3.6.15 Auto RRN Creation

Specify whether the RRN column is created automatically.

This option is set by the [3.12 Load Other File](#) options when the file has to include RRNO column and the RRNO value is automatically generated by the database when doing the SQL INSERT operation.

It is also set when creating a new LANSAs file. There is also an option to set Auto RRN on a File object from its pull down menu.

It is only enabled for PC Other Files and for LANSAs files that do not have the option set yet.

It is disabled for all LANSAs files that have the option set. That is, it cannot be unset once it has been set.

Once a file has been set Auto RRN on, LANSAs does not support setting it off again. This includes importing an older version and checking out an older version from the IBM i Master or the VCS Master.

Default = YES (selected).

[↑ 3.6 File Attributes](#)

3.6.16 Create RRNO Column

Specify whether or not RRN functionality is required for this file. It is only available if the File is a PC Other File.

- YES specifies that the table requires the X_RRNO and X_UPID columns (@@RRNO and @@UPID fields), so that crossed-update checks can be made and LANSAs RRN functionality can be used. An automatically incrementing column will be used for X_RRNO.
- NO indicates that the table does not require the X_RRNO and X_UPID columns. This option has been deprecated but is retained for backward compatibility.

Warnings

- Removing the X_RRNO and X_UPID columns will limit the functionality that can be performed on a file.

Following is a partial list of limitations:

- Any command that uses RRN functionality will not be supported (for example: WITH_RRN and RETURN_RRN parameters).
- Virtual field derivation code may not work.
- The @@UPID field will have an undefined value.
- Crossed update checking will be limited: there is a remote possibility that an update made by another user will be overwritten (for example: two batch jobs simultaneously updating the same set of rows).
- If there is no primary key on the file, UPDATE and DELETE commands are not supported.
- Most templates shipped with LANSAs will not work without these columns.
- SQL is only reusable for INSERT statements. (With Create RRNO Column, it would be reusable for UPDATE, DELETE and re-read SQL statements too.) That is, you get better performance on UPDATE and DELETE if you add the @@UPID and @@RRNO fields to the file.

Tips & Techniques

- For PC Other Files, the initial value of Create RRNO Column can be set during the load of the file. It defaults to NO for new installations. The choice selected is remembered for the next load. (Refer to [Other Data Sources Load](#)

[Options.\)](#)

Platform Considerations

- **IBM i:** This option is ignored for LANSAs files. It must be YES (selected) for Other files.

Note: With the introduction of setting *Auto RRNO* on LANSAs files, it is now mandatory to create the RRNO column on new LANSAs files. Setting Auto RRNO creates an Identity column to store the RRNO.

This provides a very fast method of generating the RRNO. It is much faster than the deprecated method of using an external file to store the next RRNO. Too many important LANSAs features are unavailable (as listed above) when the RRNO is not used. You may consider that the RRNO is an unnecessary overhead, but it is so essential to LANSAs that without it, much functionality becomes far more difficult for you to implement. In practice, both developing LANSAs and execution of LANSAs are at least as fast if not faster, with RRNO.

Also See

[3.12 Load Other File](#)

[↑ 3.6 File Attributes](#)

3.6.17 Convert Special Characters in Field Names

Specifies whether or not field names should be used as-is when creating database columns from field names.

Default = NO (unchecked/not selected) for files created from LANSAs Version 11 SP4 onwards. Files created in earlier versions default to YES (checked/selected).

Prior to LANSAs Version 11 SP4, any files compiled under Windows might create database columns that did not match the field names. For example, the SECTION field on the Demonstration file SECTAB was created as S_SECTION because SECTION was an SQL Keyword. For the same reason, the column MY@FLD would be created as MY_FLD.

From Version 11 SP4, field names in all new files will not be converted in the database.

This now allows Visual LANSAs to use files that previously would not compile on Windows and failed with error 979. For example, if you had two fields named MY@FLD and MY#FLD on a file, the file would fail to build on Windows with the message:

979 FATAL - Fields MY@FLD and MY#FLD resolve to the same SQL column name MY_FLD

The file could only be used on IBM i. Now, you can change the setting to NO (unchecked/not selected) and the file will build successfully.

Warnings

- LANSAs provides no support for changing this setting to NO (unchecked/not selected) for files that have previously compiled successfully on Windows. That is, you must only set it to unchecked/not selected from checked/selected if the file has failed to compile with error 979.
- LANSAs provides no support for changing this setting to YES (checked/selected) for files created with V11 SP4 or later.
- If you change this setting, you must also recompile and redeploy any functions or components that use SELECT_SQL against the file, and any OAMs for files that have this file as a target of Batch Control or PJFs.
- Any non-LANSAs applications that access the table should use the actual field names for the column names, and should quote these identifiers to avoid any issues.

Tips & Techniques

- Leave the setting as its default unless you have had problems with error 979 on file compile.

Platform Considerations

- IBM i: This file attribute does not apply to files in IBM i databases.

[↑ 3.6 File Attributes](#)

3.6.18 Commitment Control

Specify whether or not the file is to be placed under commitment control. For information, refer to [Commitment Control](#) in the *LANSA Application Design Guide*.

Default = NO (unchecked/not selected).

Warnings

- Using this option indicates that the file is to be placed under commitment control all the time, in all applications.

Tips & Techniques

- To selectively use (or not use) commitment control, refer to the *PGMCOMMIT / *NOPGMCOMMIT options of the [FUNCTION](#) command.
- When a file definition is first created, the default value for commitment control is set from the system definition block. Refer to [System Definition Data Areas](#) in the *LANSA for i User Guide* for more details and information about how to change the system default values.

Platform Considerations

- IBM i: Refer to the appropriate IBM supplied manuals for more details of commitment control and commitment control processing. Commitment control is a facility provided by the IBM i operating systems.

Also See

[3.6.19 Auto Commit](#)

[↑ 3.6 File Attributes](#)

3.6.19 Auto Commit

The Auto Commit option was made redundant by LANSAs release 4.0 at program change level E5.

Default = NO (unchecked/not selected).

Tips & Techniques

- To use commitment control specify COMMIT and/or ROLLBACK commands in your application.
- Generally only COMMIT commands are required.
- In the event of an application failure, ROLLBACK operations are automatically issued by the operating system when the file involved is closed.

Platform Considerations

- IBM i: Refer to the appropriate IBM supplied manuals for more details of commitment control and commitment control processing. Commitment control is a facility provided by the IBM i operating systems.

Also See

[3.6.18 Commitment Control](#)

[↑ 3.6 File Attributes](#)

3.6.20 CRTPF and CHGPF Parameter

Specify any additional command parameters that are to be used by LANSA when creating (CRTPF command) or changing (CHGPF command) the physical file.

When the file definition is set up this value is initialized to include SIZE and LVLCHK parameters which are set from the default values specified in the system definition block. Refer to [System Definition Data Areas](#) for more details.
Default = SIZE(10000 2000 3) LVLCHK(*YES)

Rules

When specifying parameters, input them exactly as would be done when entering them through the IBM i command entry facility. Use "keyword" (rather than positional) specification of any parameters required.

Parameters that can be specified include:

- EXPDATE
- MAXMBRS
- MAINT
- RECOVER
- FRCACCPH
- SIZE
- ALLOCATE
- UNIT
- FRCRATIO
- WAITFILE
- WAITRCD
- SHARE
- DLTPCT
- LVLCHK

The SHARE parameter relates to the CRTPF and CHGPF command common parameters only, it does not relate to the share an open data option. Refer to [3.6.7 Share](#).

Parameters specified are checked for validity. If invalid the screen will be re-displayed with an error message.

Platform Considerations

- IBM i: Refer to the IBM supplied manual *Control Language Reference Manual* for more details of the CRTPF and CHGPF commands and the associated common parameters.

[↑ 3.6 File Attributes](#)

3.6.21 Readonly Access

Readonly access indicates files with contents that cannot be updated through LANSAs.

The attribute applies to *Other Files* and the default is **No** (that is, updating is allowed).

It can be set to **Yes** (no updating allowed) if the *Allow read only access* option is enabled during the loading process.

It is possible that LANSAs will be unable to update a file regardless of this indicator's setting.

You can set this option for *Other Files* as described in [3.12.4 Other Data Sources Load Options](#).

3.6.22 Database File Trigger

This option enables a Database Trigger program to be generated for access by external programs.

The program will be generated using the *Database Trigger Program* name which you provide when you select the *Enable Database Triggers* option.

Default = **DB** followed by the first letters of the owning File name, for example **DBFILEM**

Rules

- The name cannot:
 - include invalid filename characters.
 - be the same as an existing file name.
 - be the same as an existing database trigger name.
- The name cannot be blank when the *Enable Database Triggers* option is selected.

Platform Considerations

- Database Triggers are only implemented for database access occurring on IBM i via Native I/O in an RDMLX partition.

Also see

in the Developer's Guide:

[File Rules & Triggers Development](#)

[LANSA Database Triggers](#)

[↑ 3.6 File Attributes](#)

3.6.23 File Description

Mandatory.

Specify the description of the physical file. This description is used within LANSAs and within the operating system to aid system users in identifying the file.

Rules

- A file description must be entered for each language defined for the partition.
- Maximum length is 40 characters.

[↑ 3.6 File Attributes](#)

3.7 The @@UPID Field in LANSAs Created Files

Whenever a physical file is created and maintained by LANSAs (rather than some OTHER system) it has an additional field placed into it. The field is called @@UPID and is defined as packed (7,0). It is always the last field in the file.

Field @@UPID is used by LANSAs to automatically check for "crossed updates". The logic to do this is very simple:

- Read the record and save the @@UPID value.
- If update required: re-read record and compare the @@UPID with saved @@UPID value. If different issue "crossed update" error message, else add 1 to @@UPID and update the file record.

When writing user application programs (in non-LANSAs applications) to write new records or update existing records in database files created by LANSAs the following is recommended:

1. Set @@UPID to 1 when writing new records.
2. Add 1 to @@UPID when updating an existing record.

This effectively emulates the logic automatically used in all LANSAs functions.

Note: COBOL programs will not like the field name @@UPID.

To solve this problem, alter the data dictionary definition of field @@UPID so that it has an associated **alias name** acceptable to COBOL (unless this has already been done).

Force recreation of all database files (that do not already have the alias name included), and then in the COBOL programs, use the COPY DD option to ensure that where a field has an alias name, it is to be used in the program in preference to its real name.

Warning: The field @@UPID should not be used at 4GL level, except where you have received specific instructions from LANSAs on how to use it.

Note: When a file contains BLOB or CLOB fields, @@UPID may be incremented multiple times for a single UPDATE command. This occurs once for the main file, and once for each BLOB or CLOB field included in the UPDATE command.

Also see

[RESET_@@UPID Built-In Function.](#)

[↑ 3. Files](#)

3.8 Batch Control

Batch control is used to define the logic by which numeric fields in one file are to be accumulated into fields in another file.

[3.8.1 Batch Control Definition](#)

Also See

[Batch Control Tab](#) in the *User Guide*.

[Batch Control Development](#) in the *Developer Guide*.

[↑ 3. Files](#)

3.8.1 Batch Control Definition

[Batch Control File Name](#)

[Batch Control Description](#)

[Batch Control Key Field](#)

[Batch Control Key Sequence](#)

[Batch Control Source Field](#)

[Batch Control Target Field](#)

[Batch Control Field Details](#)

Also See

[Batch Control Tab](#) in the *User Guide*.

[Batch Control Development](#) in the *Developer Guide*.

[↑ 3.8 Batch Control](#)

Batch Control File Name

Mandatory.

Specify the name of the physical or logical file that is to be maintained by this batch control logic. This is file that will hold the accumulated totals from the currently selected file.

Rules

- The physical or logical file specified must exist in the repository.
- The file specified cannot contain batch control logic itself.

Tips & Techniques

- If the specified file contains batch control and causes a problem, investigate moving the batch control logic from the nominated "batch control file" to this file (the file definition currently being worked with). In most cases this technique will satisfy all batch control requirements.

Also See

[Batch Control Target Field](#)

[↑ 3.8.1 Batch Control Definition](#)

Batch Control Description

Mandatory.

Specify a description that will aid other users of this file definition in identifying the purpose of the batch control logic.

Rules

- Maximum length is 40 characters.

[↑ 3.8.1 Batch Control Definition](#)

Batch Control Key Field

Mandatory.

Specify the fields or values that should be used to form the key that will be used to access the appropriate record in the file named as the "batch control file" identified by the [Batch Control File Name](#).

If field names are used, then they must be defined in this file (i.e. the file definition that is currently selected and not the file named as the "batch control" file).

Key values are checked for type and length compatibility. The entire key list supplied is checked for compatibility with the actual key(s) of the "batch control file". The key list specified can be a full or partial key to the file.

Rules

Allowable values are:

- A field that is defined in the current file definition. You must not use a virtual field.
- An alphanumeric literal (in quotes) such as 'NSW', 'BALMAIN'
- A numeric literal such as 1, 14.23, -1.141217.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.

Warnings

- A warning is issued if a partial key list is specified. The use of a partial key in this particular situation would be rare. If a warning message is issued carefully check and reconsider exactly what batch control logic has just been defined.

Tips & Techniques

- If you need to use a virtual field in the key, then code the batch control update logic into a trigger. Use of virtual fields may lead to unpredictable and/or unexpected results.

Also See

[Batch Control File Name](#)

[↑ 3.8.1 Batch Control Definition](#)

Batch Control Key Sequence

Mandatory.

Specify the sequence number for the [Batch Control Key Field](#).

Rules

- Sequence numbers must be consecutive.
- The sequence number is automatically assigned when a key is added. Sequence number is updated when the order of the keys is updated in the list of keys.

[↑ 3.8.1 Batch Control Definition](#)

Batch Control Source Field

Mandatory.

Specify from 1 to 4 fields in this file (i.e. the file definition currently selected) that are to be accumulated into the "batch control file" identified by the [Batch Control File Name](#).

At least one pair of fields (source and target) is required.

Rules

- All fields nominated must be defined in this file.
- All fields must be numeric.

Warnings

- The checking (and testing) of accumulated field precision is a user responsibility and is not performed by LANSA.

Tips & Techniques

- Note that while LANSA will check that all fields nominated exist in their respective files, and are numeric, it will not check the sizes. Thus it is possible to accumulate a field with 4 decimal positions into a field with no decimal positions. All decimal precision would be lost and the accumulation would probably be meaningless. Likewise a field with 15 significant digits could be accumulated into a field with 1 significant digit. Again the accumulation would almost certainly be meaningless.

Also See

[Batch Control Target Field](#)

[↑ 3.8.1 Batch Control Definition](#)

Batch Control Target Field

Mandatory.

Specify from 1 to 4 fields in the "batch control file" that are to hold the accumulations of the fields from this file.

At least one pair of fields (source and target) is required.

Rules

- All fields nominated must be defined in the file identified in the [Batch Control File Name](#).
- All fields must be numeric.

Warnings

- The checking (and testing) of accumulated field precision is a user responsibility and is not performed by LANSA.

Tips & Techniques

- Note that while LANSA will check that all fields nominated exist in their respective files, and are numeric, it will not check the sizes. Thus it is possible to accumulate a field with 4 decimal positions into a field with no decimal positions. All decimal precision would be lost and the accumulation would probably be meaningless. Likewise a field with 15 significant digits could be accumulated into a field with 1 significant digit. Again the accumulation would almost certainly be meaningless.

Also See

[Batch Control Source Field](#)

[↑ 3.8 Batch Control](#)

Batch Control Field Details

Display Only.

The fields details of the specified [Batch Control Key Field](#), [Batch Control Source Field](#), or [Batch Control Target Field](#) will be displayed including:

- [Description](#)
- [Type](#)
- [Length](#)
- [Decimals](#)

You can review this information to confirm field compatibility.

[↑ 3.8 Batch Control](#)

3.9 Virtual Derivation

In addition to the extended definition virtual fields (refer to [3.3 Virtual Fields in File](#)), virtual fields can be defined by entering code that is included into the I/O module.

[3.9.1 I/O Module Section](#)

[3.9.2 Virtual Code](#)

Also See

[Virtual Derivation Tab](#) in the *User Guide*.

[Virtual Fields](#) in the *LANSA for i User Guide*.

[↑ 3. Files](#)

3.9.1 I/O Module Section

Specify the sections the part of the file I/O module in which you can include user code to be used to derive virtual fields.

When LANSAs is automatically generating an I/O module, it looks for any virtual field code associated with the file and automatically includes it at the correct point.

Rules

Allowable values are:

ARRAYSPECIFICATIONS	Array specifications
CALCULATIONSAFTERINPUT	Calculations after input
CALCULATIONSBEFOREOUTPUT	Calculations before output
COMPILETIMEARRAYDATA	Compile time array data
DATASTRUCTURES	Data structures
EXTERNALFIELDRENAMES	External field renames
FILESPECIFICATIONS	File specifications
INTERNALSUBROUTINES	Internal subroutines
OUTPUTSPECIFICATIONS	Output specifications

Tips & Techniques

- The sections of the I/O module that will require code may depend upon the language being used. For example, RPG has a very structured program layout that requires code definitions in specific parts of the program.

Also See

[3.9.2 Virtual Code](#)

[↑ 3.9 Virtual Derivation](#)

3.9.2 Virtual Code

Specify the code for the virtual fields that is to be imbedded into the I/O module.

When LANSAs is automatically generating the code for an I/O module, it looks for any virtual field code associated with the file and automatically includes it at the correct point.

Rules

- RPG or C code may be entered.
- C code must have the letter C as the first character of each line.

Warnings

- There is no syntax checking performed on the code entered.

Tips & Techniques

- The sections of RPG or C code displayed describe the parts or portions of the file I/O module in which you can include user code that is to be used to derive virtual fields.
- [Triggers](#) or, for RDMLX files [3.3.6 Code Fragment](#), can be used for most complex virtual field derivation. These are much simpler than writing C or RPG code.

Platform Considerations

- IBM i: Please refer to [Examples of Virtual Fields and Derivation Code](#) in the *LANSAs for i User Guide* for specific RPG coding requirements.

Also See

[3.9.1 I/O Module Section](#)

[↑ 3.9 Virtual Derivation](#)

3.10 File Rules and Triggers

Rules and Triggers can be stored in the LANSAs Repository at both the field level and file level.

It is important to understand how the rules and triggers work at both levels and how the levels work together. Refer to [Rules and Triggers Development](#) in the *Developer Guide*.

Both field and file level details are found in [Rules and Triggers](#) in this guide.

Also see

[File Rules and Triggers Tab](#) in the *User Guide*.

[↑ 3. Files](#)

3.11 File Compile Options

Select the options which should be used when generating the Tables/Indexes/OAMs for the selected file. The options will be saved after completion of the current operation.

3.11.1 Compile only if necessary

3.11.2 Rebuild table

3.11.3 Rebuild indexes and views

3.11.4 Rebuild OAMs

3.11.5 Strip debug information

3.11.6 Keep generated source

3.11.7 Keep saved data (DAT file)

3.11.8 Drop existing tables/indexes

3.11.9 Save table data

3.11.10 Reload table data

↑ 3. Files

3.11.1 Compile only if necessary

Select this option so that only those files needing to be compiled (i.e. files with a status of rebuild required) are compiled. This is the default setting.

If it is not selected, all the selected files are forced to be compiled.

[↑ 3.11 File Compile Options](#)

3.11.2 Rebuild table

This option is only available if you have not selected the [3.11.1 Compile only if necessary](#) check box.

Select this option to force the generation and compilation of the table, indexes and OAM.

Also See

[3.11.3 Rebuild indexes and views](#)

[3.11.4 Rebuild OAMs](#)

[↑ 3.11 File Compile Options](#)

3.11.3 Rebuild indexes and views

This option is only available if you have not selected the [3.11.1 Compile only if necessary](#) check box.

Select this option to force the generation and compilation of the indexes and OAM.

Also See

[3.11.2 Rebuild table](#)

[3.11.4 Rebuild OAMs](#)

[↑ 3.11 File Compile Options](#)

3.11.4 Rebuild OAMs

This option is only available if you have not selected the [3.11 File Compile Options](#) check box.

Select this option to force the generation and compilation of the OAM.

Also See

[3.11.2 Rebuild table](#)

[3.11.3 Rebuild indexes and views](#)

[↑ 3.11 File Compile Options](#)

3.11.5 Strip debug information

Specifies whether to retain RPG or C/C++ debugging information in the compiled file.

On Windows, refer to [Producing Debug Symbols for Your LANSA Application](#) in the *Administrator's Guide* for important information that is vital to keep securely for rare but critical situations.

On the System i, this information is only required in two situations:

- When attempting to use the IBM supplied debugging aids with the compiled RPG I/O module.
- When restoring a compiled RPG Object Access Module (OAM) shipped from a System/38 onto an IBM i.

Since these situations are relatively rare, the default for this field is YES. Debugging information should be stripped.

By using this option, the size of the compiled Object Access Module (OAM) will typically be reduced by 40 - 60%. This size reduction has no bearing on execution speed, just on the size of the compiled object.

To enable RPG OAMs to be debugged, and in environments where they will be ported from System/38s to System i, this option should not be checked (selected).

[↑ 3.11 File Compile Options](#)

3.11.6 Keep generated source

Select this option to keep the generated source code. The default option is not to keep the source code.

The source code needs to be kept if the resulting objects are to be executed on a platform other than Microsoft Windows. It is also required in order to fully resolve dump files, though its possible to produce this when needed, provided the original Visual LANSA development environment is retained.

If the source code is being moved to another machine for subsequent (re)compilation, you must use this option.

The source code to be kept includes:

- The table/index/view/OAM creation code if generating a file
- The process/function/component creation code if generating a process, or function or component.
- The corresponding define and make files.

[↑ 3.11 File Compile Options](#)

3.11.7 Keep saved data (DAT file)

Select this option to keep the data that is created by the [3.11.9 Save table data](#) option.

If this option is not used, the data is only deleted if the reload of the table data was successful.

If this option is used, the data is never deleted. This may mean that large amounts of disk space are consumed.

[↑ 3.11 File Compile Options](#)

3.11.8 Drop existing tables/indexes

Drop the existing table/index before creating the new table/index definition.

Select this option when an existing table/index/view is to be generated/compiled so that the existing SQL table/index/view definition is removed before re-creation.

Note: Refer to the [3.11.9 Save table data](#) option for saving data before the table is dropped.

[↑ 3.11 File Compile Options](#)

3.11.9 Save table data

This option is only available when you have also selected the [3.11.8 Drop existing tables/indexes](#) check box.

Select this option when you wish to have the existing data in the table(s) saved before the table is deleted and recreated during a LANSa file compile.

Using this option will result in all existing data being saved to a file named ffffffff.DAT (where ffffffff is the table name) in the partition directory X_LANSa_ppp (where ppp is the partition identifier) in delimited ASCII format.

Format of the data file is:

column 1 data, column 2 data, column 3 data,

This format applies to every row in the table.

All alphanumeric data is delimited by double quotes("").

It is your responsibility to restore any data back into the table using the data saved in the ffffffff.DAT file. You may also use the [3.11.10 Reload table data](#).

[↑ 3.11 File Compile Options](#)

3.11.10 Reload table data

This option is only available when you have also selected the [3.11.8 Drop existing tables/indexes](#) check box. Selecting this box will cause the [3.11.9 Save table data](#) box to be automatically selected.

Select this option when you wish to have the existing data in the tables(s) restored after the table is deleted and recreated during a LANSAs defined file generation/creation.

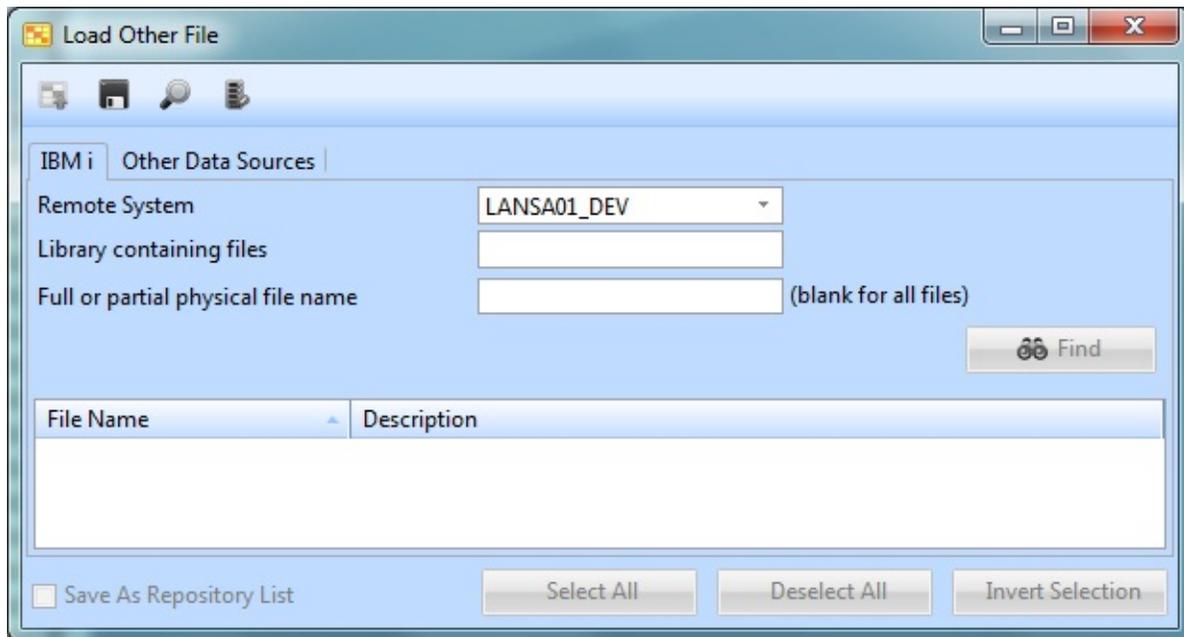
Also See

[Load Other Files](#) in the *Developer Guide*.

[↑ 3.11 File Compile Options](#)

3.12 Load Other File

To load file definitions whose definitions are maintained outside of LANSAs, choose the *New > File > Load Other File* option from the *File* menu.



The *Load Other File* dialog will be displayed. If your LANSAs installation is a Slave to an IBM i Master system or has remote IBM I system defined, an IBM I tab will be shown. Use this tab to load file definitions that reside on the selected IBM I system. An Other Data Sources tab will always be shown. Use this tab to load files from databases as defined by the ODBC Data Sources configured on your PC.

[3.12.1 Loading IBM i Files](#)

[3.12.2 Loading Files From Other Data Sources](#)

[3.12.3 IBM i Load Options](#)

[3.12.4 Other Data Sources Load Options](#)

Also See

[Load Other Files](#) in the *Developer Guide*

[↑ 3. Files](#)

3.12.1 Loading IBM i Files

In order to load IBM i file definitions, you search an IBM i library and select the file name from the list of files located. Your search will include:

Remote System

Select remote IBM i system to be searched. System must be defined as a **Remote System**.

Library containing files

Mandatory.

Specify the name of the library in which the files to be loaded reside.

Full or partial file name

Specify the full name or part of the file name to be found.

If no full or partial file name has been specified all files in the library will be listed in the search results. If the library contains many files, the search results may take some time to be displayed.

Search Results

Once the search criteria have been specified, and the Find button has been pressed, a list of matching files will be displayed.

Each file displayed in the search results list includes:

- A clickable expansion button, if the file has logical views. Clicking the button when it shows a plus sign (+) will display the file's logical views. Clicking it when a minus sign (-) is shown will hide them. Once a file has been selected, any logical views belonging to the file will be enabled and may also be selected for loading.
- A check box. Select the check box to load the file definition. Note that until a check box is checked/selected, the check boxes for any logical files belonging to the file will be disabled. The Select All, Deselect All and Invert Selection buttons at the bottom of the form can be used for managing multiple selections.
- A green tick or red cross, indicating whether or not its definition has been previously loaded into the repository.

Selected files are loaded by clicking the  Load button on the toolbar. Prior to the load being performed, the Load Options dialog will be displayed. Refer to [3.12.3 IBM i Load Options](#) for more information.

If warning or error messages are generated by the load process, a warning or error triangle will be displayed adjacent to the physical file's check box. Click on the triangle to view the messages.

Note that warning and error messages for all selected files, as well as those messages that may not be specific to a particular file, are accessible by clicking the inverted triangle that appears adjacent to the Search results text, above the list of files.

Load messages can also be displayed by clicking the  Messages button on the toolbar. From the resultant messages dialog, the text can be copied and then pasted elsewhere. This can be useful if you need to contact LANSAs Support about a loading issue.

Once a file definition has been loaded, its repository definitions can be previewed by selecting the file and clicking the  Preview button.

If no error messages have been generated by the validation process, the  Save button will be enabled. Click it to save the selected file definitions to the LANSAs repository.

Note that, by changing the selection of files in the Search results, the Save button will be disabled and the load must be performed again.

Also See

[Load Other Files](#) in the *Developer Guide*

[↑ 3.12 Load Other File](#)

3.12.2 Loading Files From Other Data Sources

In order to load file definitions, you must search existing data sources and select the file name from the list of files located. Your search will include:

Connect to Database

Mandatory.

Select the database from the list provided.

Note: Some data sources can only be searched once you have supplied a valid User ID and Password in a Connect to Database dialog.

If the MS Access Database data source is selected, a Select Database dialog will be displayed. It is similar to a standard Windows File Open dialog.

Full or partial table name

Specify the full name or part of the file name to be found. Note that this edit box is case-sensitive, and that no results will be found if the case entered does not match the case of your table names.

If no full or partial physical file name has been specified all files in the data source will be listed in the search results. If the data source contains many files, the search results may take some time to be displayed.

Search Results

Once the search criteria have been specified, and the Find button has been pressed, a list of matching tables will be displayed.

Each table displayed in the search results list includes:

- A check box. Select the check box to load the table definition. The Select All, Deselect All and Invert Selection buttons at the bottom of the form can be used for managing multiple selections.
- A green tick or red cross, indicating whether or not its definition has been previously loaded into the repository.

Selected tables are loaded by clicking the  Load button on the toolbar. Prior to the load being performed, the Load Options dialog will be displayed. Refer to [3.12.4 Other Data Sources Load Options](#) for more information.

If warning or error messages are generated by the load process, a warning or error triangle will be displayed adjacent to the table's check box. Click on the triangle to view the messages.

Note that warning and error messages for all selected tables, as well as those

messages that may not be specific to a particular table, are accessible by clicking the inverted triangle that appears adjacent to the Search results text, above the list of tables.

Load messages can also be displayed by clicking the  Messages button on the toolbar. From the resultant dialog, the text can be copied and then pasted elsewhere. This can be useful if you need to contact LANSa Support about a loading issue.

Once a table definition has been loaded, its repository definitions can be previewed by selecting the table and clicking the  Preview button.

If no error messages have been generated by the validation process, the  Save button will be enabled. Click it to save the selected table definitions to the LANSa repository.

Note that, by changing the selection of tables in the Search results, the Save button will be disabled and the load must be performed again.

Also See

[Load Other Files](#) in the *Developer Guide*

[↑ 3.12 Load Other File](#)

3.12.3 IBM i Load Options

The following load options may be specified when loading IBM i files after you select this icon  in the top right corner of the *Load Other File* list:

Enable for RDMLX

Check/select this option if the file contains RDMLX field types (refer to [What Classifies a Field as RDMLX](#) for more information) and you wish to allow the selected file(s) to be used only by RDMLX-enabled components or functions. Refer to [RDML and RDMLX Partition Concepts](#) for more information.

If this option is not checked/selected and the file(s) being loaded contain fields that are only supported by RDMLX, error messages will be generated by the load.

Submit compiles

Check/select this option if you wish to compile the loaded file definitions once they have been saved in the repository.

Suppress IOM0034

This option will set the Suppress IOM0034 attribute of all loaded file definitions accordingly.

Commitment Control

This option will set the Commitment Control attribute of all loaded file definitions accordingly.

I/O Module Library

This option will set the I/O Module Library attribute of all loaded file definitions accordingly.

[↑ 3.12 Load Other File](#)

3.12.4 Other Data Sources Load Options

 Select this icon after you have highlighted a file on the *Load Other File* import list to change the following options:

Submit compiles

Select this option if you wish to compile the loaded table definitions once they have been saved in the repository.

Allow read access only

Select this option if you do not wish any changes to be made to the table. This will automatically disable Create RRNO Column and Add Columns to Support LOBs. Any Inserts, Updates, or Deletes attempted against the file will be disallowed.

This option is displayed as [3.6.21 Readonly Access](#) on the *File Attributes* tab.

Add columns to support LOBs

Select this option if you wish to access BLOB or CLOB fields from the file the same way as they are read from a LANSAs file. Additional columns will be added to the file when it is compiled, allowing a particular file name to be associated with each BLOB or CLOB field value when it is read from the database.

Database privileges are required to modify external tables. Please refer to [Load Other Files](#) in the *Developer Guide* for more information.

Create RRNO column

Select this option if you require RRN functionality to be enabled for the loaded tables. Refer to [3.6.16 Create RRNO Column](#) for more information about this file attribute.

Database privileges are required to modify external tables. Please refer to [Load Other Files](#) in the *Developer Guide* for more information.

Suppress IOM0034

This option will set the Suppress IOM0034 attribute of all loaded table definitions accordingly.

Commitment Control

This option will set the Commitment Control attribute of all loaded table definitions accordingly.

I/O Module Library

This option will set the I/O Module Library attribute of all loaded table definitions accordingly.

Also See

[Load Other Files](#) in the *Developer Guide*.

[↑ 3.12 Load Other File](#)

4. Components

[4.1 Component Concepts](#)

[4.2 Component Definition](#)

[4.3 Component Types:](#)

[4.3.1 Form](#) [4.3.4 Visual Style](#) [4.3.7 Cursor](#)

[4.3.2 Reusable Part](#) [4.3.5 Icon](#) [4.3.8 ActiveX](#)

[4.3.3 WAM](#) [4.3.6 Bitmap](#) [4.3.9 .NET Components](#)

[4.4 Component Help Text](#)

[4.5 Component Compile Options](#)

[4.6 Technology Services](#)

Also See

[Fields](#)

[Edit Components and Functions](#) in the *User Guide*

4.1 Component Concepts

Visual LANSA extends the LANSA repository to include components in addition to fields and files. This object-oriented component model provides the foundation for user-centered, event-driven applications. However, in Visual LANSA the object-oriented paradigm is implemented in a simple way to allow you to focus on productivity and business goals.

Components are programming objects that support properties, events and methods. LANSA components are stored in the LANSA Repository. With components, you create event-driven applications.

Visual LANSA has many user definable [4.3 Component Types](#). The most common component is a [4.3.1 Form](#) which corresponds to a window of an application. [4.3.2 Reusable Part](#) parts contain controls and code which can be reused in forms (or other reusable parts).

Properties

All Visual LANSA components have properties which define their characteristics. Most properties deal with the way a component is displayed on the screen: its size, color, and whether it is visible or not. You can set the properties when you are designing your application, or programmatically when the application is running. Often you do not need to change any of the default property values.

Events

An event is a thing that happens or takes place. Typically an event is any action the user takes. The most common event is click (when the user clicks on a component with the mouse). You could for example define a button 'Print' and write code for the button's click event to print something. When the application is running, every time your user clicks on the Print button the print routine will be executed.

Methods

A method is how you tell a component to do something. For example to display a form, you use the ShowForm method:

```
INVOKE #FormB.ShowForm
```

You can also define custom methods to make a component perform an action.

Also See

[4.2 Component Definition](#)

[4.3 Component Types](#)

[4.4 Component Help Text](#)

[Edit Components and Functions](#) in the *User Guide*

[↑ 4. Components](#)

4.2 Component Definition

[4.2.1 Component Name](#)

[4.2.2 Component Identifier](#)

[4.2.3 Component Description](#)

[4.2.4 Enable Components for RDMLX](#)

[4.2.5 Framework](#)

[4.2.6 Group](#)

[4.2.9 Layout Weblet](#)

[4.2.7 Open In Editor](#)

[4.2.8 Close](#)

Also see

[4.1 Component Concepts](#)

[Edit Components and Functions](#) in the *User Guide*

[↑ 4. Components](#)

4.2.1 Component Name

Mandatory.

Specify the name of the component to be stored in the LANSAs Repository. Component names are not case sensitive. Component names are not converted to upper case characters in LANSAs.

Rules

- Must be a valid [LANSAs object name](#).

Tips & Techniques

- Define standards for component names. Different standards may apply to different component types.
- Properly defined naming standards are extremely helpful when you are sharing components with other developers and when you attempt to deploy your finished applications.

Also See

[4.2.2 Component Identifier](#)

[4.1 Component Concepts](#)

[4.3 Component Types](#)

[Edit Objects](#) in the *User Guide*

[↑ 4.2 Component Definition](#)

4.2.2 Component Identifier

Mandatory.

Specify the identifier of the component to be stored in the LANSAs Repository. Component identifiers are not case sensitive. By default, component identifiers are often converted to upper case characters in LANSAs.

Rules

- Must be a valid [LANSAs object name](#).

Tips & Techniques

- Define standards for component identifiers. Different standards may apply to different component types.
- Properly defined naming standards are extremely helpful when you are sharing components with other developers and when you attempt to deploy your finished applications.

Also See

[4.2.1 Component Name](#)

[4.1 Component Concepts](#)

[4.3 Component Types](#)

[Edit Objects](#) in the *User Guide*

[↑ 4.2 Component Definition](#)

4.2.3 Component Description

Mandatory.

- Specify the description associated with the component. The description text may be used as the default description when component information is displayed in the repository or in the finished application. If the partition is multilingual, the description specified for the default partition language will be used for other languages.

Tips & Techniques

- Use upper and lower case characters for the description.
- For a Form, the component description can be used as the default title in the window title bar.
- Define standards for component descriptions. Different standards may apply to different component types.

Also See

[4.3 Component Types](#)

[Create Components](#) in the *User Guide*

[↑ 4.2 Component Definition](#)

4.2.4 Enable Components for RDMLX

To change an existing RDML Component to an RDMLX Component, open the component in the Visual LANSA Editor, choose the *File* menu and select the *Enable for Full RDMLX* option.

This option is only available in an RDMLX Enabled Partition.

Once you select this option, the code in the component will be evaluated using the full RDMLX Language Features. Errors, if any, must be corrected before you will be able to enable it for full RDMLX

A message will be displayed when the component is saved unless the Visual LANSA environment settings have been set so no user confirmation is required.

If a component is enabled for full RDMLX, it means that it can use RDMLX objects such as fields and files.

The default value for this option is controlled in the [RDMLX Partition Settings](#).

Tips & Techniques

- It is recommended that you review the [RDML and RDMLX Partition Concepts](#) information in the Administrator Guide.

Implications:

- If no changes have been made to the code in an enabled RDMLX Component, the resulting program should be functionally equivalent to the program created by the RDML Component. However, it is your responsibility to retest the functionality of the new program.
- Performance characteristics may change and should be properly evaluated once the conversion to RDMLX has been made.

Warning

- Once a component has been enabled as a Full RDMLX Component, it cannot be changed back to an RDML Component. The code in the component would need to be copied to an RDML Component and any RDMLX features removed from the code.

[↑ 4.2 Component Definition](#)

4.2.5 Framework

Mandatory.

Specify the framework with which you want to associate the component.

Frameworks are a business-oriented grouping of items.

Rules

- You may select only one Framework to associate with the component.

Tips & Techniques

- It is easier to manage your application if you put all its forms in the same group or framework. Often you might want to create a new group for the application.
- Groups and Frameworks are defined on the IBM i and imported to Visual LANSA.

Also See

[4.2.6 Group](#)

[↑ 4.2 Component Definition](#)

[Creating Components](#)

4.2.6 Group

Specify the group or groups with which you want to associate the component. Groups are a development-oriented means of grouping similar items together.

Rules

- A group does not have to be selected.
- You may select (highlight in the list) one or more groups to associate with the component.

Tips & Techniques

- Groups and Frameworks are defined on the IBM i and imported to Visual LANSa.
- You might assign a commonly used shared component to a specific Group.
- User defined Lists are commonly used to organize application components within the editor.

Also See

[4.2.5 Framework](#)

[Create Components](#) in the *User Guide*

[↑ 4.2 Component Definition](#)

4.2.7 Open In Editor

Select this option if you want the component opened immediately in the Editor so that you can start working on it.

If you don't select (ü) this option, the component will be added to the Repository and you can open it in the Editor later.

[↑ 4.2 Component Definition](#)

4.2.8 Close

Select the *Close* option (ü) if you do not want to create another component of the same type.

If you leave this option blank, the dialog will remain open to enable you to enter the next component.

[↑ 4.2 Component Definition](#)

4.2.9 Layout Weblet

Only applies to Web Application Modules.

The name of the Layout Weblet to be used as the default layout for the WAM.

A Layout Weblet provides the basic HTML document structure (html, head, body, script, style, etc.) required by all web pages.

It is suggested that you should create, and use, a site Layout Weblet to maintain a consistent look and feel for all your company's web pages.

The Layout Weblet you choose may be:

- one of the LANSA shipped Layout Weblets
- an existing WAM Layout Weblet
- your own site Layout Weblet.

Your own site Layout Weblet can be created using the [Web Application Layout Wizard](#) or you may choose to create your own Layout Weblet.

If you leave this field blank, LANSA will use the default Layout Weblet.

LANSA will generate a default layout specific to your WAM using the nominated or default Layout Weblet.

Once a WAM is created using a specific Layout Weblet, all subsequent New WAM dialogs will default to use the same layout.

Also see

[WAM Layouts and Layout Weblets](#) in the Web Application Modules (WAMs) Guide.

[Create a WAM](#) in the *User Guide*.

[↑ 4.2 Component Definition](#)

4.3 Component Types

Visual LANSA has the following types of user definable components:

- [4.3.1 Form](#): A form corresponds to a window of an application.
- [4.3.2 Reusable Part](#): These parts contain controls and code which can be reused in forms (or other reusable parts).
- [4.3.4 Visual Style](#): This is a special kind of component which controls the appearance of individual fields, forms and controls or entire applications.
- [4.3.3 WAM](#): Web Application Modules are component-based web technology used for building LANSA applications for the Internet
- [4.3.5 Icon](#): These are images which are shared by many applications.
- [4.3.6 Bitmap](#): These are images which are shared by many applications.
- [4.3.7 Cursor](#): These are images which are shared by many applications.
- [4.3.8 ActiveX](#): This is a Microsoft ActiveX control.
- [4.3.9 .NET Components](#): Third-party reusable components that have been created using the Microsoft .NET Framework.

Platform Considerations

- **General**: Restrictions may apply to the execution of some component types on specific platforms. For example, IBM i platform does not support the execution of Forms with a visual interface; however, server-side components are supported.
- **Windows**: All component types are supported on Windows.

Also See

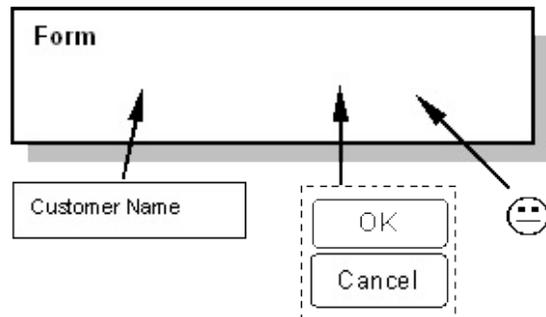
[4.1 Component Concepts](#)

[4.2 Component Definition](#)

[↑ 4. Components](#)

4.3.1 Form

A form is a component which corresponds to a window of your application when it is running. You create the interface of your application by dragging components from the repository to forms. It might include fields, buttons, lists, [4.3.2 Reusable Part](#) or other types of components. You can also put various purely visual things such as [4.3.5 Icon](#), [4.3.6 Bitmap](#) and videos on forms.



The form consists of two parts: a visual/graphical design or interface layout, and source code. A form uses RDML/RDMLX code for building the program logic to support the interface.

Typically, an application will consist of more than one form to create a "multi-form application".

Also See

[4.2 Component Definition](#)

[↑ 4.3 Component Types](#)

4.3.2 Reusable Part

Component technology is designed to provide productivity, quality and consistency gains by centering the development effort on the creation of standard, reusable, automated building blocks, called reusable parts, from which applications are assembled. As much of the maintenance is carried out at the reusable part level, the time required in testing and verifying individual applications is also greatly reduced.

A typical Visual LANSA application is built from a number of reusable parts defined in the repository, such as lists, fields and standard dialogs. You can change an individual reusable part and this change is reflected in every application that uses the component—the applications themselves do not need to be changed or recompiled. For example you could change the label of a reusable part and this change would be reflected immediately in every application where the button is used.

Reusable parts allow you to define components that can be reused in many different applications. It is like using any other component such as a push button. A reusable part presents to the outside world as:

- *a set of properties* - which you can set and get just like the properties associated with a push button.
- *a set of methods* - which you can invoke to request that the part performs some activity.
- *a set of events* - which you can monitor for so that you are notified when something significant happens within the reusable part.

Reusable Parts can be used whenever you suspect that the logic you are going to create can be simplified, standardized and reused in more than one form.

Note that you cannot execute a reusable part in isolation. It has to be imbedded in a [4.3.1 Form](#) before you can observe and test its operation.

Also See

[4.2 Component Definition](#)

[↑ 4.3 Component Types](#)

4.3.3 WAM

Web Application Modules, also called WAMs, are a component-based Web technology used for building LANSA applications for the Internet. WAM components are enabled for Full RDMLX and are defined with component properties, events and methods (PEM). The WAM Editor also includes WYSIWYG editing features for customizing any XSL presentation associated with the WAM.

WAMs use a standard XML/XSL Architecture which is open and rapidly adaptable to technology changes. By using an XML/XSL Architecture, WAMs are able to create an independence between the data specification and presentation specification. This independence is very important as new computing device and presentation formats are defined. For example, if you build a solution for only HTML, what happens when you require a cellular solution requiring WML or some other technology. The WAM architecture is able to deliver technology services for a variety of client computing devices including browsers.

Also See

[An Introduction to WAMs](#)

[4.2 Component Definition](#)

[↑ 4.3 Component Types](#)

4.3.4 Visual Style

The appearance of Visual LANSA components is controlled by visual style components. A visual style component controls the appearance of the application including colors, fonts and 3D effects.

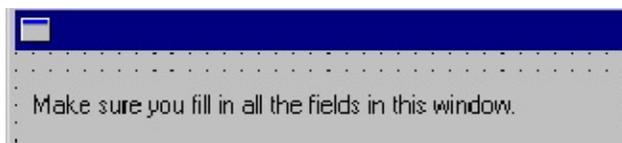
You can set one visual style to govern your entire application. This ensures absolute uniformity and makes it possible to make global changes by changing a single setting. Visual styles are multilingual so that you can define different settings for different languages in one style.

To define how items such as a text label will be displayed, LANSA supports the component property called a VisualStyle property. This property can be assigned to a Visual Style defined in the LANSA Repository.

For example, a LANSA shipped visual style VS_Norm defines the following properties for a label:

BorderStyle	3DLeft (3D effect, shadow on the left)
NormBackColor	ButtonFace (the color specified for buttons in the Windows BUTTONFACE system value, usually gray)
TextColor	WindowText (the color specified for windows text in the WINDOWTEXT system value, usually black)
Face Name	MS Sans Serif
FontSize	8 points

and it will look like this:



LANSA comes with predefined visual styles. You can use them, change them or create your own. To get the full benefit of visual styles, they should be managed centrally.

Also See

[4.2 Component Definition](#)

[↑ 4.3 Component Types](#)

4.3.5 Icon

When you want to use icons or bitmaps in many of the Visual LANSA controls such as tree and list views, forms, push-buttons and toolbar buttons you must first enroll the image in the repository as a bitmap or icon component.

Using repository-enrolled images makes development and application deployment easier because the images are managed centrally so no separate image files need to be maintained or distributed. Also, you do not need to recompile your forms when you change an image.

There is a set of icons and bitmaps supplied with LANSA (called VB_XXXXXX and VI_XXXXXX).

Visual LANSA does not provide an icon editor, but many cheap or free icon editors are available on the Web.

Libraries of icons are also accessible on the Web. Usually these libraries are free or available for a nominal charge. An example of a site that offers icon libraries: [Kira's Icon Library](#). A Web search will reveal other icon libraries.

Also See

[4.2 Component Definition](#)

[↑ 4.3 Component Types](#)

4.3.6 Bitmap

When you want to use icons or bitmaps in many of the Visual LANSA controls such as tree and list views, forms, push-buttons and toolbar buttons you must first enroll the image in the repository as a bitmap or icon component.

Using repository-enrolled images makes development and application deployment easier because the images are managed centrally so no separate image files need to be maintained or distributed. Also, you do not need to recompile your forms when you change an image.

There is a set of icons and bitmaps supplied with LANSA (called VB_XXXXXX and VI_XXXXXX).

Visual LANSA does not provide an icon editor, but many cheap or free icon editors are available on the Web.

Libraries of icons are also accessible on the Web. Usually these libraries are free or available for a nominal charge. An example of a site that offers icon libraries: [Kira's Icon Library](#). A Web search will reveal other icon libraries.

Also See

[4.2 Component Definition](#)

[↑ 4.3 Component Types](#)

4.3.7 Cursor

When you want to use specific cursor images for operations such as drag and drop, you must first enroll the image in the repository as a cursor component.

When you create cursor components, you need to specify the name of an existing .cur file as the value of the FileName property. You should be able to find .cur files in your Windows directory.

Also See

[4.2 Component Definition](#)

[↑ 4.3 Component Types](#)

4.3.8 ActiveX

ActiveX controls are third-party standard reusable components which you can use in any application that supports ActiveX technology. Visual LANSAs provides full support of ActiveX.

There are ActiveX controls for, for instance, calendars, text formatting, graphing, spell checking, advanced grids or Web browsers. You can also integrate many applications, such as the Microsoft Office Suite (Word, Excel, Powerpoint etc.) into your Visual LANSAs application using ActiveX technology.

You need to enrol ActiveX-enabled applications in the LANSAs repository as a component before you can use them from your LANSAs application.

You work with ActiveX controls the same way as you work with any other controls: by using the control's properties, methods and events. Some controls are very simple and little programming is required to use them, others provide complex functionality and may require much more coding than is needed than when using native Visual LANSAs controls.

Visual LANSAs also allows developers to expose LANSAs components as ActiveX controls on Microsoft Windows. When a Visual LANSAs component (form or reusable part) is compiled, ActiveX controls can be generated exposing nominated properties, events and methods. These features will be exposed via attributes in the components code.

Also See

[4.2 Component Definition](#)

[Using ActiveX Controls](#) in the *Developers Guide*.

[↑ 4.3 Component Types](#)

4.3.9 .NET Components

.NET Components are third-party reusable components which have been created using the Microsoft .NET Framework. .NET Components may be components or controls that you have created yourself, purchased from a third-party, or standard controls in the Microsoft .NET Framework.

Examples of .NET Framework components are calendars, graphing, spell checking, advanced grids and browsers.

LANSA provides support for the use of .NET Framework components and controls in Visual LANSA projects.

Also See

[4.2 Component Definition](#)

[Before You Decide to Use a .NET Component](#) in the *Developers Guide*.

[↑ 4.3 Component Types](#)

4.4 Component Help Text

Component help text can be stored within the component definition (refer to [DEFINE_COM](#)) or it can be stored in the repository as it with field, function and process help.

Help text is information that is displayed to the user when application requests help (using the Help key or equivalent request). Help text can be entered for each language specified in the partition.

Generally Help text has the following characteristics:

- It is free format. No restrictions usually exist on the content or format of Help text.
- It relates directly to the action the user was taking at the time the Help was requested. Usually an overall description of the component that the user is using is presented.
- Help text may also include special [Help Text Enhancement & Substitution Values](#).

LANSA automatically controls the handling of the Help processing in applications. LANSAs will automatically determine the type of Help that is required (field, component, process or function) and automatically display the associated Help text (if any exists).

You can also build your own application level help that integrates online help and your application documentation. Refer to SET 230 example [Using CHM files for Online Help Text](#).

Also See

In the *Developer Guide*: [Repository Help Editor](#)

In this guide: [Substitution/Control Values](#)
[Substitution/Control Values - Visual LANSAs Only](#)
[Help Text Attributes](#).

[↑ 4. Components](#)

4.5 Component Compile Options

Select the options that should be used when generating and/or compiling the selected components. The options will be saved after completion of the current operation.

[4.5.1 Compile Component only if necessary](#)

[4.5.2 Keep Generated Source](#)

[4.5.3 Debug Enabled](#)

[4.5.4 Web Application Module Options](#)

[↑ 4. Components](#)

4.5.1 Compile Component only if necessary

Select this option so that only those components that need to be compiled are compiled. This is the default setting.

If it is not selected, all the selected components are compiled.

[↑ 4.5 Component Compile Options](#)

4.5.2 Keep Generated Source

Select this option to keep the generated source code. The default option is not to keep the source code.

The source code needs to be kept if the resulting objects are to be executed on a platform other than Microsoft Windows. It is also required in order to fully resolve dump files, though its possible to produce this when needed, provided the original Visual LANSA development environment is retained.

If the source code is being moved to another machine for subsequent (re)compilation, you must use this option.

The source code to be kept includes:

- The table/index/view/OAM creation code if generating a file
- The process/function/component creation code if generating a process, or function or component.
- The corresponding define and make files.

[↑ 4.5 Component Compile Options](#)

4.5.3 Debug Enabled

Select this option to compile the objects with debug information.

If you use this option, the object can be debugged using the LANSAs debug tools. The executable objects will be slightly larger than compiling without debug.

Refer to [Producing Debug Symbols for Your LANSAs Application](#) in the *Administrator's Guide* for important information that is vital to keep securely for rare but critical situations.

[↑ 4.5 Component Compile Options](#)

4.5.4 Web Application Module Options

When compiling a Web Application Module or when one or more of the selected objects to compile is a Web Application Module, the Generate XML/XSL checkbox is enabled.

Generated code can only execute from the Web.

Also See

[Generate XSL](#)

[Technology Services](#)

[↑ 4.5 Component Compile Options](#)

Generate XSL

Select this option if you want to generate both the Web application Module's input XML and XSL stylesheets. If you do not select this option, only the input XML is generated for the Web Application Module's webroutines.

When you select to generate XSL, you have the option to do this for all webroutines or only for new webroutines which do not have existing XSL stylesheets. The default option is to generate only for new webroutines.

[↑ 4.5.4 Web Application Module Options](#)

Technology Services

Select the Technology Services that you want to generate. An XSL stylesheet will be created for the selected Technology Services. If generating XSL, at least one Technology Service must be selected.

[↑ 4.5.4 Web Application Module Options](#)

4.6 Technology Services

Technology Services is a presentation or XML format used by a WAM to interact with a user agent or other XML-aware application.

LANSAs provides Technology Services for XHTML and Pocket PC HTML. A Technology Service is uniquely identified by the combination of the Technology Service provider and the Technology Service name, for example:

LANSAs:XHTML.

To create a Technology Service, refer to [Edit Technology Service Definitions](#) in the *User Guide*.

[4.6.1 Technology Service Name](#)

[4.6.2 Technology Service Provider Name](#)

[4.6.3 Technology Service Caption](#)

[4.6.4 Technology Service Description](#)

[4.6.5 Technology Service Properties](#)

Also See

[Web Application Module Guide](#)

[↑ 4. Components](#)

4.6.1 Technology Service Name

Mandatory.

The name of the presentation or XML format. It normally matches the name of the standardized format (For example, XHTML or WML).

A Technology Service is uniquely identified by the combination of the Technology Service provider and the Technology Service name, for example: LANSA:XHTML.

Rules

- Maximum 10 characters.

Also see

[4.6.2 Technology Service Provider Name](#)

[↑ 4.6 Technology Services](#)

4.6.2 Technology Service Provider Name

Mandatory.

The name of the party who provided this Technology Service.

Any Technology Services supplied by LANSA will be defined under the Technology Service Provider: LANSA. If you intend to create your own Technology Services, it is recommended that you use a unique Technology Service Provider name to avoid any clashes with LANSA definitions in future releases.

For example, LANSA provides an XHTML Technology Service. The combination of Technology Service Provider and Technology Service is used to uniquely identify the Technology Service as LANSA:XHTML. If you need to define your own variation of XHTML, you use your own Technology Service Provider and a new Technology Service associated with this Provider, so you may have your version of XHTML identified as <Provider>:XHTML.

Rules

- Maximum 10 characters.

[↑ 4.6 Technology Services](#)

4.6.3 Technology Service Caption

Mandatory.

A short description by which the Technology Service can be known.

This description can be entered via the Details tab after you have created the Technology Service.

Rules

- Maximum 25 characters.

[↑ 4.6 Technology Services](#)

4.6.4 Technology Service Description

Mandatory.

Specify the description of the Technology Service.

[↑ 4.6 Technology Services](#)

4.6.5 Technology Service Properties

[MIME Type](#)

[Document Extension](#)

[Edit Numeric](#)

[Designable in Editor](#)

[Maximum Footprint](#)

[Device Skin Image](#)

[Script Location](#)

[Style Location](#)

Also See

[Web Application Module Guide](#)

[↑ 4.6 Technology Services](#)

MIME Type

Mandatory.

Specify the MIME type to include in the HTTP header for an outgoing response.

Rules

Allowable values are:

text/html Text or HTML documents

[↑ 4.6.5 Technology Service Properties](#)

Document Extension

Mandatory.

Specify the document extension associated with documents (stream files) for the Technology Service. For example, HTML pages are identified by the extension .html.

Rules

- Maximum 10 characters.

[↑ 4.6.5 Technology Service Properties](#)

Edit Numeric

Mandatory.

Specify whether the Technology Service should edit numeric values or not. Presentation oriented formats normally edit numeric fields. If your Technology Service is to be consumed by an application, you would not want edit symbols in your numeric values.

[↑ 4.6.5 Technology Service Properties](#)

Designable in Editor

Mandatory.

This property is used to determine if you can visually design (use drag-and-drop) the Technology Service. This is normally set to "Yes" only for Technology services shipped by LANSA. If you are creating a customized version of any of Technology Services shipped by LANSA, you may set this property to "Yes" but you are responsible for compatibility.

[↑ 4.6.5 Technology Service Properties](#)

Maximum Footprint

Optional.

This property is used in the Design view to present the actual screen size of the Technology Service end-user interface. Most devices, such as PDAs, have small screen sizes which you must allow for when designing presentation in WAM Editor.

[↑ 4.6.5 Technology Service Properties](#)

Device Skin Image

Optional.

This property is used to display a device background image in the Design view. This file can be in most image file formats and it is loaded from the web server images virtual directory.

[↑ 4.6.5 Technology Service Properties](#)

Script Location

Mandatory.

The default location for web external resources of type script. The WAM guide describes these locations.

[↑ 4.6.5 Technology Service Properties](#)

Style Location

Mandatory.

The default location for web external resources of type style. The WAM guide describes these locations.

[↑ 4.6.5 Technology Service Properties](#)

5. Webllets

Webllets are reusable components for common HTML functions and can simply be dragged and dropped into your web pages.

[5.1 Weblet Name](#)

[5.4 Layout Weblet](#)

[5.2 Weblet Description](#)

[5.5 Webroutine Service Name](#)

[5.3 Weblet Group](#)

[↑ 5. Webllets](#)

5.1 Weblet Name

Mandatory.

Specify the name of the weblet to be stored in the LANSAs Repository. Weblet names are not case sensitive.

Rules

- Must be a valid [LANSA Object Name](#).

Warnings

- Avoid the use of names like SQLxxx, as this may cause problems when used in functions that use SQL (Structured Query Language) facilities. (For example, Command SELECT_SQL.)

Tips & Techniques

- Name can be either lower case or upper case.

[↑ 5. Weblets](#)

5.2 Weblet Description

Mandatory.

- Specify the description associated with the weblet. The description text may be used as the default description when weblet information is displayed in the repository or in the finished application.
If the partition is multilingual, the initial description is copied as the description for all other languages in the partition. You must change each description if you wish it to be in a different language.

Tips & Techniques

- You can use upper and lower case characters for the description.

Also See

[Create Weblet](#) in the *User Guide*

[↑ 5. Weblets](#)

5.3 Weblet Group

Optional.

- Weblet Groups allow you to group similar weblets together.
Nominate the weblet group to which the weblet will be added. If the weblet group does not already exist, the group will be created. If no group is nominated, the weblet is added to the 'Unassigned' group.

[↑ 5. Weblets](#)

5.4 Layout Weblet

Optional.

- Select this option if you want to create a layout weblet. A layout weblet provides the basic HTML document structure (html, head, body, script, style, etc.) required by all web pages.

Also See

[Create Weblet](#) in the *User Guide*

[↑ 5. Weblets](#)

5.5 Webroutine Service Name

A WAM's webroutine can uniquely either be identified by its name and the name of the WAM it belongs to. It can also be uniquely identified by one name, called the Service Name.

When launching a webroutine in a browser use the following URL:

http://localhost/cgi-bin/lansaweb?webapp=<WAM name>+webrtn=<WEBROUTINE name>+ml=<TS name>+part=<PARTITION name>+lang=<LANGUAGE name>

If a service name has been associated with a Webroutine, the unique service name can be specified:

http://localhost/cgi-bin/lansaweb?srve=<Service name>+ml=<TS name>+part=<PARTITION name>+lang=<LANGUAGE name>

The Service Name has to be unique in the partition it is used in.

Using a WEBROUTINE Service Name provides greater flexibility when deploying WAM applications. For example, it allows applications to be re-deployed to a different Partition, WAM or WEBROUTINE without having to modify any external URL references to it.

[↑ 5. Weblets](#)

6. Processes and Functions

Process Topics:

- [6.1 Process Definition](#)
- [6.2 Function Control Table](#)
- [6.3 Special Entries](#)
- [6.4 Attached Processes/Functions](#)
- [6.5 Action Bar Table](#)
- [6.6 Process Help Text](#)
- [6.9 Process/Function Compile Options](#)

Function Topics:

- [6.7 Function Definition](#)
- [6.8 Function Help Text](#)

Also See

In the User Guide:

[Editing Processes](#)

[Editing Components and Functions.](#)

In the Developer Guide:

[Developing with Processes and Functions.](#)

6.1 Process Definition

[6.1.1 Process Name](#)

[6.1.2 Process Identifier](#)

[6.1.3 Process Description](#)

[6.1.4 Menu Style](#)

[6.1.5 Anticipated Usage](#)

[6.1.6 Optimize for remote communications](#)

[6.1.7 Enable for Web](#)

[6.1.8 Generate XML](#)

Also See

[Editing Processes](#) in the *User Guide*.

[Creating Processes](#) in the *User Guide*.

[What is a Process](#) in the *Developer Guide*.

[Developing with Processes and Functions](#) in the *Developer Guide*.

[↑ 6. Processes and Functions](#)

6.1.1 Process Name

Mandatory.

Specify the name of the process to be stored in the repository.

Rules

- Must be a valid [LANSA object name](#).

Warnings

- Please refer to [LANSA object name](#).

Platform Considerations

- Please refer to [LANSA object name](#).
- A process name must be unique within the entire LANSAs partition.

Tips & Techniques

- It is recommended that a naming standard be developed for process names.

Also see

[Creating Processes](#) in the *User Guide*.

[6.1.2 Process Identifier](#)

[↑ 6.1 Process Definition](#)

6.1.2 Process Identifier

Mandatory.

Specify the identifier of the process to be stored in the repository.

Rules

- Must be a valid [LANSA object name](#).

Warnings

- Please refer to process details described in [LANSA object name](#).

Platform Considerations

- Please refer to process details described in [LANSA object name](#).
- A process identifier must be unique within the entire LANSAs partition.

Tips & Techniques

- It is recommended that a naming standard be developed for process identifiers.

Also see

[Creating Processes](#) in the *User Guide*.

[6.1.1 Process Name](#)

[↑ 6.1 Process Definition](#)

6.1.3 Process Description

Mandatory.

Specify the description to be associated with the process. The description aids other users of this process in identifying what it can be used for. If the partition is multilingual, the description specified for the default partition language will be used for other languages.

Rules

- A description must be entered for each language defined for the partition.
- Maximum length is 40 characters.

Also see

[Creating Processes](#) in the *User Guide*.

[↑ 6.1 Process Definition](#)

6.1.4 Menu Style

Mandatory. Default= SAA/CUA

Specify the menu style of the process that is to be created.

Rules

Allowable values are:

SAA/CUA All menus and screen formats used by this process and any of its associated functions are to conform to the SAA (Systems Application Architecture) and/or CUA (Common User Access) standards defined for the partition in which the process is being defined. Refer to *Partition Definitions* in [SAA/CUA Implementation](#) in the *LANSA Application Design Guide* for more details of what the SAA/CUA standards are for a partition and how they apply.

Action Bar The process is to act as an "Action bar" as defined by the CUA (Common User Access) standards defined by IBM and for this partition.

Warning

If using Action Bar, then the following prerequisites must be satisfied:

- The current partition must be SAA/CUA enabled.
- You must read all relevant information in the [LANSA Application Design Guide](#) and in the IBM supplied *CUA 1989 Basic Interface Design Guide*.
- You must be totally committed to the CUA 1989 standard for the "look" and "feel" of your application software.

[↑ 6.1 Process Definition](#)

6.1.5 Anticipated Usage

Mandatory. Default=Light.

Specify the amount of usage of the process which is anticipated.

Rules

Allowable values are:

LIGHT Anticipated usage is LIGHT. The process will not be used repeatedly and continuously. Most processes are considered to be LIGHT usage.

HEAVY Anticipated usage is heavy. The process will be used repeatedly and continuously. This option is normally only used in repetitive data entry applications.

Platform Considerations

- IBM i: In technical terms the Anticipated Usage value indicates whether or not the RPG programs created for the functions in this process should set on the LR (last record) indicator and close all files when terminating.
- IBM i: The Anticipated Usage value can be changed dynamically (without having to recompile any programs) so it may be worthwhile experimenting with it to modify system performance/throughput.

[↑ 6.1 Process Definition](#)

6.1.6 Optimize for remote communications

Default=No

Specify whether remote communications are optimized for all functions within this process.

Platform Considerations

- IBM i: Please refer to [Miscellaneous Process Details Maintenance](#) in the *LANSAs for i User Guide* for information about optimizing remote communications.

[↑ 6.1 Process Definition](#)

6.1.7 Enable for Web

Default=No.

[↑ 6.1 Process Definition](#)

6.1.8 Generate XML

Default=No.

[↑ 6.1 Process Definition](#)

6.2 Function Control Table

[6.2.1 Function Description](#)

[6.2.2 Display on Menu](#)

[6.2.3 Menu Sequence](#)

[6.2.4 Next Function](#)

[6.2.5 Allowable Next Function\(s\)](#)

Also See

[6. Processes and Functions](#) [Function Tab](#) in the *User Guide*.

[Function Control Table Concepts](#) in the *Developer Guide*.

[↑ 6. Processes and Functions](#)

6.2.1 Function Description

Mandatory.

Specify the description associated with the function. The description of the function will be displayed on the process menu and in the LANSAs Repository.

Rules

- A file description must be entered for each language defined for the partition.
- Maximum length is 40 characters.

Tips & Techniques

- This option is often used to change the description of the function in the LANSAs Repository.
- Since this value will appear on the process menu, it is recommended that you use upper and lower case characters.
- There is no need to use blanks to centre the description as this is done automatically on the process menu.

Also See

[Function Control Table Concepts](#) in the *Developer Guide*.

[6.7 Function Definition](#)

[↑ 6.2 Function Control Table](#)

6.2.2 Display on Menu

Mandatory. Default=Yes

Specify whether or not this function should be displayed (and therefore be accessible from) the process's main menu.

Also See

[Function Control Table Concepts](#) in the *Developer Guide*.

[↑ 6.2 Function Control Table](#)

6.2.3 Menu Sequence

Mandatory. Default=Next sequential number.

Specify the order of the functions on the Process Menu. If the function is to be displayed on the process menu then the order in which it appears on the process menu can be governed by modifying the sequencing.

Tips & Techniques

- Example: If the process contains FUNC1, FUNC2 and FUNC3. If you wish to modify the sequencing such that FUNC3 appears before FUNC2, then a sequence number which is less than the sequence number for FUNC2 is entered in this entry field. This will then cause the function sequencing to be FUNC1, FUNC3 and then FUNC2.

Also See

[Function Control Table Concepts](#) in the *Developer Guide*.

[↑ 6.2 Function Control Table](#)

6.2.4 Next Function

Mandatory. Default=MENU

Specify the function that is "usually" invoked after this function has been completed.

Rules

- Allowable values are any function defined in the process or any of the "reserved" function names, except for *ANY which makes no sense in this context. Refer to Allowable Next Function(s).

Also See

[Function Control Table Concepts](#) in the *Developer Guide*.

[↑ 6.2 Function Control Table](#)

6.2.5 Allowable Next Function(s)

Mandatory. Default=*ANY

Specify up to 20 functions that are allowed to be invoked after this function has been completed.

Rules

Allowable values include all functions in the process as well as the following reserved words:

Name	Reserved meaning / description
-------------	---------------------------------------

*ANY	Any function name
MENU	Display process main menu
EXIT	Exit from LANSAP
HELP	Display process HELP text
SELECT	Select next function from list of allowable function
EOJ	End all batch processing
ERROR	Abort process with an error
RETRN	Return control to calling process or function

Tips & Techniques

- The use of special value *ANY is recommended if any function can be invoked rather than listing all function names.
- If *ANY is used, it should be the only entry in the list.
- If *ANY is not used, ensure that the Next Function is included into the list.
- Lists that do not include "reserved" names EXIT and MENU are effectively disabling the use of the EXIT and MENU function keys. The EXIT and MENU function keys are processed by simulating the entry of "next functions" of EXIT and MENU respectively.

Also See

[Function Control Table Concepts](#) in the *Developer Guide*.

↑ 6.2 Function Control Table

6.3 Special Entries

[6.3.1 Description](#)

[6.3.2 Sequence](#)

[6.3.3 Runtime Prompt](#)

[6.3.4 Command](#)

Also See

[Special Entries Tab](#) in the *User Guide*.

[Special Entries Concepts](#) in the *Developer Guide*.

[↑ 6. Processes and Functions](#)

6.3.1 Description

Specify a user description of the purpose of the special command that has been defined to LANSAs. This description will appear on the process menu.

Rules

- Must be a valid operating system command for the platform where the process is being executed.

Also See

[Special Entries Concepts](#) in the *Developer Guide*.

[↑ 6.3 Special Entries](#)

6.3.2 Sequence

Mandatory. Default=Next available sequential number.

Specify the order in which the special menu entry will appear on the selected processes menu.

Tips & Techniques

- The ordering of the menu entries can be resequenced by varying the value in this field.

Also See

[Special Entries Concepts](#) in the *Developer Guide*.

[↑ 6.3 Special Entries](#)

6.3.3 Runtime Prompt

Specify whether or not the command should be "prompted" when it is used from the process's main menu.

This value used with IBM i only. If this option is selected, the command will be prompted before executing. When used, the prompt is the standard IBM CL command style prompting. Refer to the appropriate IBM supplied manual for more details.

Also See

[Special Entries Concepts](#) in the *Developer Guide*.

[↑ 6.3 Special Entries](#)

6.3.4 Command

Mandatory.

Specify the operating system command that is to be executed when the entry is chosen from the process's main menu.

Rules

- The command should be entered exactly as it would be on the command entry display screen.
- You are responsible to ensure that a proper path and parameters have been specified to enable the command to execute properly.

Platform Considerations

- The use of special entries may not be portable between operating systems. For example, IBM i CL commands cannot be executed on a Windows platform. Use caution with special entries if you are building applications for more than one platform.

Also See

[Special Entries Concepts](#) in the *Developer Guide*.

[↑ 6.3 Special Entries](#)

6.4 Attached Processes/Functions

[6.4.1 Process Name](#)

[6.4.2 Function Name](#)

[6.4.3 Sequence](#)

[6.4.4 Process Parameters](#)

Also See

[Attachments Tab](#) in the *User Guide*.

[Attached Processes/Functions Concepts](#) in the *Developer Guide*.

[↑ 6. Processes and Functions](#)

6.4.1 Process Name

Mandatory.

Specify the name of the processes to be attached.

Rules

- The name of the process to be attached must be nominated when attaching a process or when directly attaching a single function.
- Note that the process nominated does not have to exist. If it does not currently exist a warning message will be issued.

Tips & Techniques

- Multiple processes may be attached to a single process.
- Multiple functions (from other processes) may be directly attached to a process.
- A process (or any of its associated functions) may be attached to itself.
- If a process B is attached to process A, then it is possible to attach process A to process B.
- Processes can be built into a "hierarchy" by using this facility. There is no limit to the "depth" of the hierarchy that can be defined, but when actually using a process the "depth" being used must not exceed 9 processes. If it does an error message will be issued indicating that it is not possible to go any deeper in the process "hierarchy" and that the required process should be accessed via a different route.

Also See

[Attached Processes/Functions Concepts](#) in the *Developer Guide*.

[↑ 6.4 Attached Processes/Functions](#)

6.4.2 Function Name

Mandatory.

Specify the name of the function that is to be directly attached to the current process or indicates that all functions in the process should be attached.

Rules

- To directly attach a function to the process specify the name of the function.
- To attach all functions (i.e.: attach the entire process) specify *ALL as the attached function name.
- Note that the function nominated does not have to exist. If it does not currently exist, a warning message will be issued.

Also See

[Attached Processes/Functions Concepts](#) in the *Developer Guide*.

[↑ 6.4 Attached Processes/Functions](#)

6.4.3 Sequence

Mandatory.

Specify the sequence number to nominate the relative order in which the attached process or function should be displayed on the process menu.

Rules

- Enter a number in the range 1 - 999 that indicates the required display order of the process or function relative to other attached processes or functions.

Also See

[Attached Processes/Functions Concepts](#) in the *Developer Guide*.

[↑ 6.4 Attached Processes/Functions](#)

6.4.4 Process Parameters

Process Parameters should not be used. They exist only for backward compatibility.

Symbolic Name

Mandatory.

The process parameter symbolic name in the form *UPnn where "nn" is the parameter number in the range 01 to 10.

This name allows the parameter to be easily accessed by the RDML commands associated with a function. Generally a parameter's symbolic name can be used anywhere in an RDML command that a normal field name or literal value could be used.

Sequence

Mandatory. Default=Next sequential number.

The sequence number to nominate the relative order in which the parameter is stored.

Sequence numbers are consecutive and must be in the range from 1 to 10.

Data Type

Mandatory. Default=Alpha.

The type of parameter that is to be defined.

Allowable values are:

Alpha The parameter is to be alphanumeric.

Numeric The parameter is to be numeric. If this option is used the parameter is in fact defined as a packed variable as this format is easiest to pass.

Length

Mandatory. Default=256

The length for type Alpha parameters or the total number of digits (including decimals) for Numeric parameters.

For type Alpha parameters the length specified must be in the range 1 to 256.

For type Numeric parameters the total number of digits must be in the range 1 to 15 and not less than the number of decimal positions specified.

Decimals

Default=0.

The number of decimals for Data Type of Numeric parameters only in the range 0 to 9 and less than or equal to the total number of digits specified.

Description

Mandatory.

A short description that is to be associated with the parameter. If it is necessary for LANSAs to display a data entry screen for specification of parameter values, this description will be displayed. A brief description of every process parameter that is defined must be supplied.

Also See

[Parameters Tab](#) in the *User Guide*

[Process Parameter Concepts](#) in the *Developer Guide*.

[↑ 6.4 Attached Processes/Functions](#)

6.5 Action Bar Table

[6.5.1 Action Bar Item Description](#)

[6.5.2 AB\\$OPT](#)

[6.5.3 Action Bar Item Sequence](#)

[6.5.4 Pull Down Item Description](#)

[6.5.5 Accelerator Key](#)

[6.5.6 PD\\$OPT](#)

[6.5.7 Pull Down Item Sequence](#)

[6.5.8 Initially Enabled](#)

[6.5.9 Association Type](#)

Also See

[Action Bar Table Tab](#) in the *User Guide*.

[Action Bar Concepts](#) in the *Developer Guide*.

[↑ 6. Processes and Functions](#)

6.5.1 Action Bar Item Description

Mandatory.

Specify the text that is to appear in the action bar to identify this action bar choice.

Rules

- Text must be entered for each language defined for the partition.

Tips & Techniques

- Try to use just one word.
- Use upper and lower case characters.
- Support for bidirectional and DBCS languages is provided.
- Conform to the CUA 1989 guidelines.
- Help option is automatic. You do not have to define it.

Also See

[Action Bar Concepts](#)

[↑ 6.5 Action Bar Table](#)

6.5.2 AB\$OPT

Mandatory.

Specify a value that allows the function to determine exactly which action bar choice was used to cause the function to be invoked. The value you specify here is placed into field AB\$OPT when this menu option is used. This field is accessible to RDML functions.

Rules

- Value specified should be unique within this action bar.
- It is an alphanumeric value.
- Do not use values CUR or ALL as they are reserved to mean "current" and "all" in the [SET_ACTION_BAR](#) Built-In Function.

Warnings

- If AB\$OPT is not already defined in the Repository, define it as alphanumeric (length 3).

Tips & Techniques

- One RDML function handling multiple action bar choices can have good performance implications. Refer to the [Sample Program: All 3 Functions in One Program](#) in the *LANSA Application Design Guide*.
- Standards for AB\$OPT values should be established.

[↑ 6.5 Action Bar Table](#)

6.5.3 Action Bar Item Sequence

Mandatory. Default=Next sequential number.

Specify the number associated with the pull down choice. This value is generally only used to reorder action bar choices.

Rules

- Must be in the range 1 to 18 and unique within the action bar.

[↑ 6.5 Action Bar Table](#)

6.5.4 Pull Down Item Description

Mandatory.

Specify the text that is to appear in the pull down to identify this pull down choice.

Rules

- Text must be entered for each language defined for the partition.

Tips & Techniques

- Use upper and lower case characters.
- Support for bidirectional and DBCS languages is provided.
- Conform to the CUA 1989 guidelines.
- Include "Fnn" to identify accelerator keys (where required).
- Include "..." ellipses for resulting pop-ups (where required).
- Help pull downs are automatic. You do not have to define them.

[↑ 6.5 Action Bar Table](#)

6.5.5 Accelerator Key

Specify the accelerator key that is to be associated with this pull down choice.

Rules

- Allowable values are F1 to F24, or "No Accelerator key".
- Avoid conflicts with other key assignments. This is not checked.

Tips & Techniques

- Avoid overuse. They will confuse users and complicate the system.
- Conform to the CUA 1989 guidelines.
- Key will be activated on any panel showing this action bar.

Platform Considerations

- IBM i: Can be changed dynamically without requiring recompilation.

[↑ 6.5 Action Bar Table](#)

6.5.6 PD\$OPT

Mandatory.

Specify the value that allows the function to decide exactly which pull down choice was used to cause it to be invoked. The value specified here is placed into field PD\$OPT when this pull down choice is used. The field is accessible to RDML functions.

Rules

- Value specified should be unique within this pull down, and preferably, within the entire action bar.
- It is an alphanumeric value.
- Do not use values CUR or ALL, as they are reserved to mean "current" and "all" in the [SET_ACTION_BAR](#) Built-In Function.

Warnings

- If PD\$OPT is not already defined in the Repository, define it as alphanumeric (length 3).

Tips & Techniques

- One RDML function handling multiple pull down choices can have good performance implications. Refer to the [Sample Program: All 3 Functions in One Program](#) in the *LANSA Application Design Guide*.
- Standards for PD\$OPT values should be established.

[↑ 6.5 Action Bar Table](#)

6.5.7 Pull Down Item Sequence

Mandatory. Default=Next consecutive number.

Specify the position of this option in the menu. Use this field to change the relative order of fields.

Rules

- Must be consecutive.

[↑ 6.5 Action Bar Table](#)

6.5.8 Initially Enabled

Mandatory. Default=Yes.

Specify whether or not this pull down choice is to be made available on the initial invocation of the action bar. Unavailable pull down choices are shown in blue and have their associated selection numbers replaced by an "*".

Tips & Techniques

- RDML program access to make pull down choices available/ unavailable is provide by the [SET_ACTION_BAR](#) Built-In Function.

[↑ 6.5 Action Bar Table](#)

6.5.9 Association Type

Mandatory.

Specifies which function/process/special entry is to be invoked when this menu option is selected.

Rules

Allowable values are:

- [Function Name](#)
- [Attachment](#)
- [Special Entry](#)
- [Undefined](#)

[↑ 6.5 Action Bar Table](#)

Function Name

Specify the RDML function that belongs to the current process.

Rules

- Function specified must exist within the current process.

[↑ 6.5.9 Association Type](#)

Attachment

Specify the "Attached Process" indicating a process that has been attached to the current process or the "Attached Function" indicating a function that has been attached to the current process from another process.

Rules

- Attached process or attached function specified must be defined as an attachment within the current process definition.

Tips & Techniques

- Attached processes may cause a menu to appear if they are menu style (i.e.: SAA/CUA) and thus do not have a menu bar or another menu bar to appear if they are menu bar style. This facility can be used to build up a "hierarchy" of menu bars in an acceptable manner.
- Attached functions may cause another menu bar to appear. This is the menu bar associated with the process to which they belong, not the current process's menu bar. This can be confusing to end users in some situations and should be carefully controlled or avoided.

[↑ 6.5.9 Association Type](#)

Special Entry

Specify a special entry that has been defined within the current process.

Rules

- Special entry specified must be defined within the current process definition.

[↑ 6.5.9 Association Type](#)

Undefined

Specify the association is not currently defined.

[↑ 6.5.9 Association Type](#)

6.6 Process Help Text

Help text is information that is displayed to the user when application requests help (using the Help key or equivalent request). Help text for processes is stored in the LANSAs Repository. This help text is automatically available from the process menu and can also be accessed when a function within the process is executed. Function Help Text can also be accessed. Help text can be entered for each language specified in the partition.

Generally Help text has the following characteristics:

- It is free format. No restrictions usually exist on the content or format of Help text.
- It relates directly to the action the user was taking at the time the Help was requested. Usually the process or function that the user is using is explained in some detail.
- Help text may also include special [Help Text Enhancement & Substitution Values](#).

LANSAs automatically controls the handling of the Help processing in applications. LANSAs will automatically determine the type of Help that is required (field, component, process or function) and automatically display the associated Help text (if any exists).

LANSAs does not automatically create the free format Help text that is associated with the processes or functions. LANSAs can dynamically, and in the correct language, create the Help text associated with a field from the repository and the rules that it contains. You can turn off this automatic field level help text feature: globally, by field, or precede it with your own Help text.

Also See

[6.8 Function Help Text](#)

In the *Developer Guide*:

[Repository Help Editor](#).

In the *User Guide*

[Repository Help Tab](#)

In this guide:

[Substitution/Control Values](#)

[Substitution/Control Values - Visual LANSAs Only](#)

[Help Text Attributes](#)

[Field Help Text.](#)

[↑ 6. Processes and Functions](#)

6.7 Function Definition

[6.7.1 Function Name](#)

[6.7.2 Function Identifier](#)

[6.7.3 Function Description](#)

[6.7.4 Template](#)

[6.7.5 Enable Functions for RDMLX](#)

Also See

[What is a Function](#) in the *Developer Guide*

[Creating Functions](#) in the *User Guide*.

[Developing with Processes and Functions](#) in the *Developer Guide*

[Getting Started with Function Development](#) in the *Developer Guide*.

[↑ 6. Processes and Functions](#)

6.7.1 Function Name

Mandatory.

Specify the name that is to be assigned to the new function.

Rules

- Must be a valid [LANSA object name](#).

Platform Considerations

- **Refer to** [LANSA object name](#)

Also See

[What is a Function](#) in the *Developer Guide*

[Creating Functions](#) in the *User Guide*

[6.7.2 Function Identifier](#)

[↑ 6.7 Function Definition](#)

6.7.2 Function Identifier

Mandatory.

Specify the identifier that is to be assigned to the new function.

Rules

- Must be a valid [LANSA object name](#).

Platform Considerations

- **IBM i:** A function identifier must be unique within the process it is created. It is possible to have two functions with the same identifier in a single partition if they are located in different processes.
- **Windows:** Function identifier must be unique in the partition. All functions must be defined as type *DIRECT.
- Please refer to function details described in [LANSA object name](#)

Warnings

Certain function identifiers are "reserved" for use by LANSAs and cannot be specified as a valid function identifier. These are:

Name	Reserved Meaning / Description
MENU	Display process main menu
EXIT	Exit from LANSAs
HELP	Display process HELP text
SELECT	Select next function from list of allowable function
EOJ	End all batch processing
ERROR	Abort process with an error
RETRN	Return control to calling process or function
*ANY	Any function identifier

For more details of why these names are "reserved" refer to the Function Control Table section.

Also See

[What is a Function](#) in the *Developer Guide*

[Creating Functions](#) in the *User Guide*

[6.7.1 Function Name](#)

[↑ 6.7 Function Definition](#)

6.7.3 Function Description

Mandatory.

Specify the description that is to be associated with the function. The description of the function will be displayed on the process menu and in the LANSAs Repository. If the partition is multilingual, the description specified for the default partition language will be used for other languages.

Rules

- Maximum length is 40 characters.

Tips & Techniques

- Function descriptions can be changed by altering the Function Control Table of the process.
- Since this value will appear on the process menu, it is recommended that upper and lower case characters are used.
- There is no need to use blanks to centre the description as this is done automatically on the process menu.

Also See

[Creating Functions](#)

[↑ 6.7 Function Definition](#)

6.7.4 Template

Default=No template selected.

Specify the name of the application template to be used to create the RDML in the function.

Note: You must open the function in the editor to allow the template to execute. Refer to [Creating Functions](#).

Tips & Techniques

Following are tips for using LANSAs templates:

- Make extensive use of the HELP function key and take your time. When using a template for the first time, read all the HELP panels associated with the template, especially the examples. Get a good idea of what the template will do and what the template won't do before you use it.
- Most templates work by using a "question and answer" session. So after selecting a template to be used, a question will most likely appear in the pop up window area.
- If no question appears, and the source is re-displayed, you can assume that the template has generated your RDML program without having to ask you any questions at all. Some very simple templates work this way (e.g.: the one that generates a basic program layout).
- When a question appears, read the question very carefully. Next, read any additional prompting information very carefully. And finally, if you still have any doubts, use the HELP function key.
- Follow instructions exactly. The concept of an application template is to generate RDML programs for you in a very quick and very consistent matter. The generalized nature of the Application Template facility and its ability to be site definable, mean that absolutely precise validation of your answers is often not practicable. Incorrect answers will cause no real problems, other than to cause RDML code that will either not compile, or not execute correctly, to be generated. This is not a real problem, but it means that you will most probably have to then fix it by manual editing of the RDML code.
- Most templates allow you to "back up" to a previously answered question and change your answer. To do this use the Cancel function key. Do not do this if the prompt indicates you should not do it.

Also See

[Getting Started with Function Development](#) in the *Developer Guide*.

[Using Application Templates](#) in the *Developer Guide*.

[Creating Functions](#) in the *User Guide*.

[↑ 6.7 Function Definition](#)

6.7.5 Enable Functions for RDMLX

To change an existing RDML Function to an RDMLX Function, open the function in the Visual LANSA Editor, choose the *File* menu and select the *Enable for Full RDMLX* option.

This option is only available in a RDMLX Enabled Partition.

A Function must be *Enabled for Full RDMLX* in order to use other RDMLX objects or to use RDMLX commands. You cannot use RDMLX Fields or RDMLX Files unless the function is enabled for RDMLX. RDMLX Functions can interact with Components.

The default value for this option is controlled in the [RDMLX Partition Settings](#).

Tips & Techniques

- Once you select this option, the code in the component will be evaluated using the full RDMLX Language Features. If it contains any errors, you must correct them before you can save it as an RDMLX function.

RDMLX Language features include:

- use of intrinsic field methods
 - use of function libraries
 - use of expressions in many parameters and properties
 - enhanced RDMLX command support such as assignment statements
 - simplified RDML statements due to the removal of quotes.
- It is recommended that you review the [RDML and RDMLX Partition Concepts](#) information in the Administrator Guide.

Implications:

- RDMLX Functions do not support 5250 interfaces. If you enable an RDML Function to use RDMLX field types, this new RDMLX Function can no longer directly support a 5250 interface. You must call an RDML Function to perform any screen interactions.
- Performance characteristics may change and should be properly evaluated once the conversion to RDMLX has been made.
- If no changes have been made to the code in an enabled RDMLX Function, the resulting program should be functionally equivalent to the program created by the RDML Function. However, it is your responsibility to retest the functionality of the new program.

Warning

- All editing must be performed using Visual LANSAs. RDMLX Functions cannot be edited from LANSAs for i. (LANSAs for i does not support development in RDMLX Partitions.)
- Once a function is enabled for RDMLX, it cannot be changed back. You can create a new RDML Function and code can be copied and pasted back into the RDML Function. Code will be syntax checked for compatibility with the RDML Function.

[↑ 6.7 Function Definition](#)

6.8 Function Help Text

Help text is information that is displayed to the user when the application requests help (using the Help key or equivalent request). Help text for functions is stored in the LANSAs Repository. This help text is automatically available as function level context sensitive help text when a function is executed. Process Help Text can also be accessed. Help text can be entered for each language specified in the partition.

Generally Help text has the following characteristics:

- It is free format. No restrictions usually exist on the content or format of Help text.
- It relates directly to the action the user was taking at the time the Help was requested. Usually the process or function that the user is using is explained in some detail.
- Help text may also include special [Help Text Enhancement & Substitution Values](#).

LANSAs automatically controls the handling of the Help processing in applications. LANSAs will automatically determine the type of Help that is required (field, component, process or function) and automatically display the associated Help text (if any exists).

LANSAs does not automatically create the free format Help text that is associated with the processes or functions. LANSAs can dynamically, and in the correct language, create the Help text associated with a field from the repository and the rules that it contains. You can turn off this automatic field level help text feature: globally, by field, or precede it with your own text.

Also See

In the *Developer Guide*:

[Repository Help Editor](#)

In the *User Guide*

[Repository Help Tab](#)

In this guide:

[Substitution/Control Values](#)

[Substitution/Control Values - Visual LANSAs Only](#)

[Help Text Attributes](#)

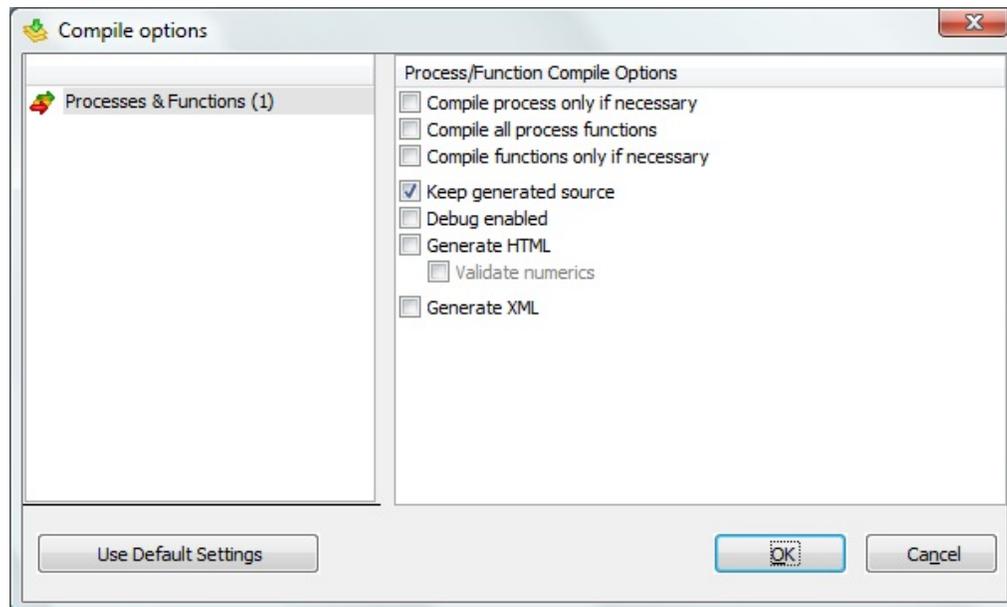
Field Help Text

↑ 6. Processes and Functions

6.9 Process/Function Compile Options

Select the options that are to be used when generating and/or compiling the selected processes or functions. The options will be saved after the operation has been completed.

The options displayed will depend on the object selected for compilation.



6.9.1 [Compile Process only if necessary](#)

6.9.2 [Compile All Process Functions](#)

6.9.3 [Compile Functions Only if Necessary](#)

6.9.4 [Keep Generated Source](#)

6.9.5 [Debug Enabled](#)

6.9.6 [Generate HTML](#)

6.9.7 [Generate XML](#)

6.9.8 [Use Default Settings](#)

[↑ 6. Processes and Functions](#)

6.9.1 Compile Process only if necessary

Select this option so that only those processes that need to be compiled are compiled. This is the default setting.

If it is not selected, all the selected processes are compiled.

[↑ 6.9 Process/Function Compile Options](#)

6.9.2 Compile All Process Functions

Select this option so that all of the functions of the selected processes will be compiled as well. This is the default setting.

If it is not selected, the functions are not compiled – only the processes are compiled.

[↑ 6.9 Process/Function Compile Options](#)

6.9.3 Compile Functions Only if Necessary

This option is only active if Compile Functions has been selected.

Select this option so that only those functions that need to be compiled are compiled. This is the default setting.

If it is not selected, all the selected functions are compiled.

[↑ 6.9 Process/Function Compile Options](#)

6.9.4 Keep Generated Source

Select this option to keep the generated source code. The default option is not to keep the source code.

The source code needs to be kept if the resulting objects are to be executed on a platform other than Microsoft Windows. It is also required in order to fully resolve dump files, though its possible to produce this when needed, provided the original Visual LANSA development environment is retained.

If the source code is being moved to another machine for subsequent (re)compilation, you must use this option.

You may also be requested to keep the generated source for problem resolution.

The source code to be kept includes:

- The table/index/view/OAM creation code if generating a file
- The process/function/component creation code if generating a process, or function or component.
- The corresponding define and make files.

[↑ 6.9 Process/Function Compile Options](#)

6.9.5 Debug Enabled

Select this option to compile the objects with debug information.

If you use this option, the object can be debugged using the LANSAs debug tools. The executable objects will be slightly larger than compiling without debug.

Refer to [Producing Debug Symbols for Your LANSAs Application](#) in the *Administrator's Guide* for important information that is vital to keep securely for rare but critical situations.

[↑ 6.9 Process/Function Compile Options](#)

6.9.6 Generate HTML

Specifies if HTML pages should be generated for DISPLAY/REQUEST/POP_UP commands when this function(s) is compiled. This option will only be available if the process is web enabled.

Default value is YES. Allowable values are :

YES Generate HTML pages for each DISPLAY/REQUEST/POP_UP command in each function that is compiled.

NO Do not generate HTML pages for each DISPLAY/REQUEST/POP_UP command in each function that is compiled.

This value should be used with caution. It is generally only used when changes to the function have not included modification of any DISPLAY/REQUEST/POP_UP command. This allows HTML pages that have been modified using an HTML editor to be preserved.

Also See

[Validate Numeric Values](#)

[↑ 6.9 Process/Function Compile Options](#)

Validate Numeric Values

Specifies if input numeric fields should be validated via JavaScript according to the allowable number of digits before and after the decimal point. This option is only applicable if the *Generate HTML pages* option is **YES**.

Allowable values are:

YES Generate JavaScript function calls for input numeric fields for each function being compiled.

NO Do not generate JavaScript function calls for input numeric fields for each function being compiled.

[↑ 6.9.6 Generate HTML](#)

6.9.7 Generate XML

Specifies if XML should be generated for DISPLAY/REQUEST/POP_UP commands when this function(s) is compiled. This option will only appear if the process is XML enabled.

Default value is YES. Allowable values are:

YES Generate XML for each DISPLAY/REQUEST/POP_UP command in each function that is compiled.

NO Do not generate XML for each DISPLAY/REQUEST/POP_UP command in each function that is compiled.

This value should be used with caution. It is generally only used when changes to the function have not included modification of any DISPLAY/REQUEST/POP_UP command. This allows XML that has been modified using the XML editor to be preserved.

[↑ 6.9 Process/Function Compile Options](#)

6.9.8 Use Default Settings

Select this button to reset all options to the shipped defaults.

[↑ 6.9 Process/Function Compile Options](#)

7. RDML Commands

Go to [RDML Commands List](#) for a summary of the commands.

The following commands make up the complete LANSa RDML programming language. Provided for each command is:

- A description
- References to related commands
- A syntax diagram
- Explanation of each parameter
- Warnings /Comments
- Examples.

For further information regarding the use and format of these commands and their parameters, refer to [RDML Command Parameters](#).

7.1 ABORT

The ABORT command is used to cause an executing RDML program to end immediately and optionally issue an error message. Ending a function via an ABORT command is considered to be an "abnormal" end and the entire process is canceled by LANSAs. For the implications of commitment control refer to [Commitment Control](#) in the *LANSAs for i User Guide*.

Also See

[7.1.1 ABORT Parameters](#)

[7.1.2 ABORT Examples](#)

[7.6 CALL](#) (see the IF_ERROR parameter)

[7.46 EXIT](#)

[7.69 MENU](#)

[7.81 RETURN](#)

Optional

```
ABORT ----- MSGTXT -----*NONE -----
---->
      'message text'

>-- MSGID ----- *NONE ----->
      message identifier

>-- MSGF ----- *NONE ----->
      message file library name

>-- MSGDTA ----- substitution variables -----|
      | expandable group expression |
      ----- 20 max -----
```

7.1.1 ABORT Parameters

MSGTXT

MSGID

MSGF

MSGDTA

MSGTXT

Allows up to 80 characters of message text to be specified. This text will be displayed when the function ends as an error message. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID/MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be issued when the function ends. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this:

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA("BOLTS  " #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.1.2 ABORT Examples

[Aborting with Simple Text](#)

[Aborting with Dynamically Constructed Text](#)

[Aborting with a Substituted Variable Message](#)

[Aborting with a Multilingual Text](#)

[Trapping an Abort](#)

Aborting with Simple Text

This command aborts a function and causes an error message to be displayed:

```
ABORT MSGTXT('Unable to locate system definition record')
```

Aborting with Dynamically Constructed Text

This subroutine dynamically constructs the error message that the ABORT command displays:

```
SUBROUTINE NAME(ABORT) PARMs((#MSGTXT1 *RECEIVED) (#MSG  
DEFINE FIELD(#MSGTXT1) TYPE(*CHAR) LENGTH(40) DECIMALS(0)  
DEFINE FIELD(#MSGTXT2) REFFLD(#MSGTXT1)  
DEFINE FIELD(#MSGTXT3) REFFLD(#MSGTXT1)  
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132) DECIMALS(0)  
USE BUILTIN(BCONCAT) WITH_ARGS(#MSGTXT1 #MSGTXT2 #MSG  
ABORT MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)  
ENDROUTINE
```

It may be used in fatal error situations like this:

```
EXECUTE SUBROUTINE(ABORT) WITH_PARMs('Employee' #EMPNO
```

Or this:

```
EXECUTE SUBROUTINE(ABORT) WITH_PARMs(#DEPARTMENT 'is inva
```

Aborting with a Substituted Variable Message

The aborting message wording can also be defined in a message file and the details substituted as variables at the time of the fatal error. A message file, e.g. MYMSGF is created and a message definition with an ID of MSG0001 is added to the file. The First Level Message Text is 'Employee &1 &2 &3 is not valid

for this tax operation because their salary of &4 is too high.' Then the Message Data Field Formats are defined like this:

- *CHAR length 5
- *CHAR length 20
- *CHAR length 20
- *DEC length 11 decimals 2

Then, the abort is given as:

```
DEFINE FIELD(#SALRY_CAP) REFFLD(#SALARY) EDIT_CODE(3) D
REQUEST FIELDS(#SALRY_CAP)
SELECT FIELDS(#EMPNO #GIVENAME #SURNAME #SALARY) FRO
IF COND('#SALARY > #SALRY_CAP')
ABORT MSGID(MSG0001) MSGF(MYMSGF) MSGDTA(#EMPNO #GI
ENDIF
ENDSELECT
MESSAGE MSGTXT('All Employees are OK') TYPE(*WINDOW) LOCAL
```

Aborting with a Multilingual Text

In multilingual applications you sometimes need to issue fatal error messages that contain *MTXT variables as their message text. This subroutine shows a way of doing this.

```
SUBROUTINE NAME(ABORT) PARMs((#MSGDTA *RECEIVED))
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132) DECIMALS
ABORT MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDROUTINE
```

It may be used in fatal error situations like this:

```
EXECUTE SUBROUTINE(ABORT) WITH_PARMs(*MTXTABORT_MES
```

Or like this:

```
EXECUTE SUBROUTINE(ABORT) WITH_PARMs(*MTXTABORT_EM
```

Trapping an Abort

The execution of an ABORT command in a called function can be detected and trapped by the calling function in this way:

```
CALL PROCESS(*DIRECT) FUNCTION(MYFUNC) IF_ERROR(ERR)
```

```
RETURN  
ERR: MESSAGE MSGTXT('MYFUNC has ended with in error') TYPE(*W  
RETURN
```

If the function MYFUNC fails, control is passed to the ERR label (note that the IF_ERROR parameter logic may be triggered for many reasons other than the execution of a ABORT command).

7.2 ADD_ENTRY

The ADD_ENTRY command is used to add a new entry to a list.

The list may be a browse list (used for displaying information at a workstation) or a working list (used to store information within a program).

Refer to the DEF_LIST command for more details of lists and list processing.

Also See

[7.2.1 ADD_ENTRY Parameters](#)

[7.2.2 ADD_ENTRY Comments / Warnings](#)

[7.2.3 ADD_ENTRY Examples](#)

[7.12 CLR_LIST](#)

[7.23 DEF_LIST](#)

[7.28 DLT_ENTRY](#)

[7.89 SORT_LIST](#)

[7.95 UPD_ENTRY](#)

Optional

```
ADD_ENTRY ---- TO_LIST ----- *FIRST -----
---->
      list name

>-- SET_SELECT --- *YES ----->
      *NO

>-- WITH_MODE ---- *CURRENT -----
-->
      *ADD
      *CHANGE
      *DELETE
      *DISPLAY
      field name
>-- AFTER----- *END -----|
      *START
      numeric value or field name
```


7.2.1 ADD_ENTRY Parameters

TO_LIST

SET_SELECT

WITH_MODE

AFTER

TO_LIST

Specifies the name of the list to which the new entry should be added.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which may be a browse or a working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

SET_SELECT

Specifies whether or not any fields in the list that have special attribute *SELECT should be set to blanks before the new entry is added to the list. Refer to the DEF_LIST command for more details.

This parameter is only valid for browse list processing. It is ignored for working list processing.

WITH_MODE

Specifies the mode to be set for the entry being added. This overrides the mode that has been set by the SET_MODE command (refer to the SET_MODE command).

The default is *CURRENT which uses the current mode that has been set by the SET_MODE command. Other allowable values are *ADD, *CHANGE, *DELETE and *DISPLAY. A user field name may also be specified, and must be alphanumeric with a length of 3, and must contain one of the values "ADD", "CHG", "DLT" or "DIS".

This parameter is only valid for browse list processing. It is ignored for working list processing.

AFTER

Specifies the position in the list where the entry is to be added.

The default value of *END specifies that the entry will be added at the end of

the list. This is the only value that will be accepted for a browse list.

The other special value of *START specifies that the entry will be added at the beginning of the list, before the current first entry. It is equivalent to specifying the numeric value of 0. This value is only valid for a working list.

A numeric value or field name specifies the number of the entry after which the new entry will be added. Specifying 0 is equivalent to the special value of *START. Apart from 0, the entry number specified must exist in the list when the ADD_ENTRY is executed. A numeric value or field name is only valid for a working list.

7.2.2 ADD_ENTRY Comments / Warnings

- ADD_ENTRY is a "mode sensitive" command when being used with a browse list. For details, refer to [RDML Screen Modes and Mode Sensitive Commands](#).
- Use of the AFTER parameter specifying anything other than *END may incur a performance penalty because of the underlying implementation of working lists. Heavy use should be benchmarked with realistically sized data sets before being put into a production environment. Possible design alternatives include replacement of the working list by a keyed work file and construction of a second working list from the first.
- Use of the AFTER parameter specifying anything other than *END also means that the entry number of all entries succeeding the newly added entry are incremented by 1. This may cause problems where the entry number of a particular entry is assumed to remain static, for example where 'pointers' to working list entries are used and also where the entries in a list are processed in a loop other than SELECTLIST/ENDSELECT.

7.2.3 ADD_ENTRY Examples

[Defining, Adding Entries To and Displaying a Browse List](#)

[Filling a Browse List with Information from a Database File](#)

[Adding Entries to the Beginning or End of a Browse List](#)

[Sorting Entries in Browse Lists by Using a Working List](#)

Defining, Adding Entries To and Displaying a Browse List

To define, add two entries to and display a browse list named #EMPBROWSE you would use these commands like this:

```
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #GIVENAME #SU
```

```
CHANGE #EMPNO 'A0001'
```

```
CHANGE #GIVENAME 'JOHN'
```

```
CHANGE #SURNAME 'SMITH'
```

```
ADD_ENTRY TO_LIST(#EMPBROWSE)
```

```
CHANGE #EMPNO 'A0002'
```

```
CHANGE #GIVENAME 'MARY'
```

```
CHANGE #SURNAME 'BROWN'
```

```
ADD_ENTRY TO_LIST(#EMPBROWSE)
```

```
DISPLAY BROWSELIST(#EMPBROWSE)
```

Filling a Browse List with Information from a Database File

This example asks the user to input a department code (eg: ADM, MKT, etc).

All employees that work in the specified are then added to a browse list named #EMPBROWSE. The resulting browse list then displayed to the user:

```
DEF_LIST NAME(#EMPBROWSE) FIELDS(#SECTION #EMPNO #SURNAME
```

```
BEGIN_LOOP
```

```
REQUEST FIELDS(#DEPARTMENT) BROWSELIST(#EMPBROWSE)
```

```
CLR_LIST NAMED(#EMPBROWSE)
```

```
SELECT FIELDS(#EMPBROWSE) FROM_FILE(PSLMST1) WITH_KEY
```

```
ADD_ENTRY TO_LIST(#EMPBROWSE)
```

```
ENDSELECT
```

```
END_LOOP
```

Adding Entries to the Beginning or End of a Browse List

This example asks the user to input sets of employee details.

Each set of employee details is added to a browse list.

These employee details may be added to the start or the end of the browse list.

Since you can't add entries to the start of a browse list a working list is used to hold the details and a browse list to display them:

```
DEFINE FIELD(#WHERE) TYPE(*CHAR) LENGTH(1) LABEL('Top or B')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #SURNAME #GIVENAME)
DEF_LIST NAME(#EMPWORKNG) FIELDS(#EMPNO #SURNAME #GIVENAME)

BEGIN_LOOP
REQUEST FIELDS(#WHERE #EMPNO #SURNAME #GIVENAME) BRCT
IF COND('#WHERE = B')
ADD_ENTRY TO_LIST(#EMPWORKNG)
ELSE
ADD_ENTRY TO_LIST(#EMPWORKNG) AFTER(*START)
ENDIF

EXECUTE SUBROUTINE(VISIBLE)
END_LOOP
```

This subroutine is used to copy the contents of the working list (#EMPWORKNG) to the visible browse list # EMPBROWSE:

```
SUBROUTINE NAME(VISIBLE)
CLR_LIST #EMPBROWSE
SELECTLIST NAMED(#EMPWORKNG)
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
ENDROUTINE
```

Sorting Entries in Browse Lists by Using a Working List

This example asks the user to input sets of employee details.

Each set of employee details is added to a browse list.

Function keys are provided to allow the browse list to be sorted in various ways.

Since you can't sort entries in a browse list a working list is used to hold the sorted details and a browse list to display in sorted order:

```
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #SURNAME #GIV
DEF_LIST NAME(#EMPWORKNG) FIELDS(#EMPNO #SURNAME #GI
```

```
DEFINE FIELD(#UK_EMPNO) REFFLD(#IO$KEY) DEFAULT("09")
DEFINE FIELD(#UK_SURNME) REFFLD(#IO$KEY) DEFAULT("10")
DEFINE FIELD(#UK_GVNNME) REFFLD(#IO$KEY) DEFAULT("11")
```

```
BEGIN_LOOP
REQUEST FIELDS(#EMPNO #SURNAME #GIVENAME) BROWSELIST
(10 'SURNAME' *NEXT)(11 'GIVNAME '))
```

```
CASE OF_FIELD(#IO$KEY)
WHEN VALUE_IS('= #UK_EMPNO')
SORT_LIST NAMED(#EMPWORKNG) BY_FIELDS(#EMPNO)
WHEN VALUE_IS('= #UK_SURNME')
SORT_LIST NAMED(#EMPWORKNG) BY_FIELDS(#SURNAME)
WHEN VALUE_IS('= #UK_GVNNME')
SORT_LIST NAMED(#EMPWORKNG) BY_FIELDS(#GIVENAME)
OTHERWISE
ADD_ENTRY TO_LIST(#EMPWORKNG)
ENDCASE
```

```
EXECUTE SUBROUTINE(VISIBLE)
END_LOOP
```

```
SUBROUTINE NAME(VISIBLE)
CLR_LIST NAMED(#EMPBROWSE)
SELECTLIST NAMED(#EMPWORKNG)
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
ENDROUTINE
RETURN
```

7.3 BEGIN_LOOP

The BEGIN_LOOP command is used in conjunction with the END_LOOP command to form a processing loop.

The loop formed is repeated the number of times specified by the FROM, TO and STEP parameters. Optionally a field can be nominated in the USING parameter which contains the value of the current iteration of the loop.

Refer to the END_LOOP command for more details and examples.

Also See

[7.3.1 BEGIN_LOOP Parameters](#)

[7.3.2 BEGIN_LOOP Examples](#)

```
BEGIN_LOOP --- USING ----- *INTERNAL -----  
----->  
                field name  
  
>--- FROM ----- 1 ----->  
                numeric value or field name  
  
>--- TO ----- 9999999 ----->  
                numeric value or field name  
  
>--- STEP ----- 1 -----|  
                non-zero numeric value (+ or -)
```

7.3.1 BEGIN_LOOP Parameters

USING

FROM

TO

STEP

USING

Optionally specifies the name of a field that is to contain the value of the current iteration of the loop.

*INTERNAL, which is the default value, indicates that no user field is to contain the current iteration value. LANSAs to create an internal field that is not accessible to user RDML program logic.

Otherwise a field name may be specified. Any field specified must be of type numeric and must be defined in the LANSAs data dictionary or in the function. In addition the field must contain enough digits to hold the maximum iteration value expected in the loop. This is not checked by LANSAs.

FROM

Specifies the start value for the first loop iteration. If this parameter is omitted, value 1 is assumed. Specify either a numeric literal or the name of a numeric field that contains the value.

TO

Specifies the final value for the last loop iteration. If this parameter is omitted, value 9999999 is assumed. Specify either a numeric literal or the name of a numeric field that contains the value.

STEP

Specifies the value by which the loop iteration counter should be incremented after each loop iteration. If this parameter is omitted, value 1 is assumed. Specify any non-zero integer for this parameter.

7.3.2 BEGIN_LOOP Examples

Example 1: Use the BEGIN_LOOP / END_LOOP commands to insert records into a file until the user uses the EXIT or menu function key:

```
GROUP_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDL1
BEGIN_LOOP
REQUEST FIELDS(#CUSTOMER) EXIT_KEY(*YES *EXIT) MENU_KI
INSERT FIELDS(#CUSTOMER) TO_FILE(CUSMST)
VAL_ERROR(*LASTDIS)
CHANGE FIELD(#CUSTOMER) TO(*DEFAULT)
END_LOOP
```

Example 2: Use the BEGIN_LOOP / END_LOOP commands to reference entries 1 through 10 of a working list called #LIST:

```
BEGIN_LOOP USING(#I) FROM(1) TO(10)
GET_ENTRY NUMBER(#I) FROM_LIST(#LIST)
END_LOOP
```

Example 3: Use the BEGIN_LOOP / END_LOOP commands to reference entries 10 through 1 (i.e.: backwards) of a working list called #LIST:

```
BEGIN_LOOP USING(#I) FROM(10) TO(1) STEP(-1)
GET_ENTRY NUMBER(#I) FROM_LIST(#LIST)
END_LOOP
```

Example 4: Use the BEGIN_LOOP / END_LOOP commands to reference even numbered entries between 1 and 100 in a working list called #LISTF:

```
BEGIN_LOOP USING(#I) FROM(2) TO(100) STEP(2)
GET_ENTRY NUMBER(#I) FROM_LIST(#LIST)
END_LOOP
```

Example 5: Use the BEGIN_LOOP / END_LOOP commands to reference odd numbered entries between 1 and 100 in a working list called #LIST:

```
BEGIN_LOOP USING(#I) FROM(1) TO(99) STEP(2)
GET_ENTRY NUMBER(#I) FROM_LIST(#LIST)
END_LOOP
```

7.4 BEGINCHECK

The BEGINCHECK command is used to specify the beginning of a block of validation checks. This command is always used in conjunction with the ENDCHECK command.

A field that is to contain an ongoing count of the number of errors that occur within the validation block may also be specified on this command.

Refer to the ENDCHECK command for more details and examples of both commands.

Also See

[7.4.1 BEGINCHECK Parameters](#)

[7.4.2 BEGINCHECK Examples](#)

[7.7 CALLCHECK](#)

[7.14 CONDCHECK](#)

[7.16 DATECHECK](#)

[7.35 ENDCHECK](#)

[7.48 FILECHECK](#)

[7.78 RANGECHECK](#)

[7.99 VALUECHECK](#)

Optional

```
BEGINCHECK --- KEEP_COUNT --- *NONE -----  
-----|  
                field name
```

7.4.1 BEGINCHECK Parameters

KEEP_COUNT

Optionally specifies the name of a field that contains an ongoing count of the number of errors detected within a BEGINCHECK/ENDCHECK validation block.

Specify the name of a numeric field that is to contain the ongoing error count. The field must be defined in the LANSAs data dictionary or in this function.

*NONE is the default value and indicates that no error counter field is required for this validation block.

Note that while LANSAs will automatically increment the field every time an error is detected within a validation block, it does **not ever set or reset the field to zero**. This is the responsibility of the programmer.

7.4.2 BEGINCHECK Examples

[Structuring Functions for Inline Validation](#)

[Structuring Functions to Use a Validation Subroutine](#)

[Using the BEGINCHECK/ENDCHECK Commands for Inline Validation](#)

[Using the BEGINCHECK/ENDCHECK Commands for Validation with a Subroutine](#)

[Structuring Functions to Validate Header Fields and Multiple Details Input Via a Browse List](#)

[Example of Validating Header Fields and Multiple Details Input Via a Browse List](#)

Structuring Functions for Inline Validation

Typically functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for inline validation like this:

```
BEGIN_LOOP
REQUEST  << INPUT >>
BEGINCHECK
*        << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK
*        << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is passed back to the REQUEST command. This happens because of the default IF_ERROR(*LASTDIS) parameter on the ENDCHECK command.

Structuring Functions to Use a Validation Subroutine

Typically functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for subroutine validation like this:

```
DEFINE  FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND  NAME(*NOERRORS) COND('#ERRORCNT = 0')
```

```

BEGIN_LOOP
DUNTIL COND(*NOERRORS)
REQUEST << INPUT >>
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
* << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP

```

```

SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
* << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE

```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is returned to the main function loop with #ERRORCNT > 0.

Using the BEGINCHECK/ENDCHECK Commands for Inline Validation

This example demonstrates how to use the BEGINCHECK/ENDCHECK commands (and the other validation commands) within the main program block to verify a set of input details.

```

DEFINE FIELD(#NEWSALARY) REFFLD(#SALARY) LABEL('New Sal
DEFINE FIELD(#TOTSALARY) REFFLD(#SALARY) DESC('Total Salar
DEFINE FIELD(#BUDGET) REFFLD(#SALARY) LABEL('Budget') DES
GROUP_BY NAME(#XG_DTails) FIELDS(#DEPARTMENT #EMPNO #SU
DEF_LIST NAME(#EMPBROWSE) FIELDS(#XG_DTails)

```

```

BEGIN_LOOP
REQUEST FIELDS(#XG_DTails #BUDGET) BROWSELIST(#EMPBR
CHANGE FIELD(#TOTSALARY) TO(*DEFAULT)
SELECT FIELDS(#SALARY) FROM_FILE(PSLMST1) WITH_KEY(#DE
CHANGE FIELD(#TOTSALARY) TO(#TOTSALARY + #SALARY')
ENDSELECT

```

```

BEGINCHECK

```

```

CALLCHECK FIELD(#STARTDTE) BY_CALLING(WORKDAY) PROG_1
CONDCHECK FIELD(#NEWSALARY) COND('(#NEWSALARY + #TOTS
DATECHECK FIELD(#STARTDTE) IN_FORMAT(*DDMMYY) BEFORE(
FILECHECK FIELD(#EMPNO) USING_FILE(PSLMST) FOUND(*ERROR
RANGECHECK FIELD(#EMPNO) RANGE((A0000 A9999)) MSGTXT('Em
VALUECHECK FIELD(#DEPARTMENT) WITH_LIST(ADM AUD FLT GAC)
ENDCHECK

```

```

ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP

```

If any of the input values causes a validation command to give an error the message defined with that command is issued and program control returns to the last screen displayed. In this case the last screen displayed is the REQUEST screen.

Using the BEGINCHECK/ENDCHECK Commands for Validation with a Subroutine

This example demonstrates how to use the BEGINCHECK/ENDCHECK commands inside a subroutine to check that entered details for a new employee conform to a set of validations before being accepted for further processing.

After the user enters the requested details the VALIDATE subroutine is called. It checks that all the values comply with the various validations. If this is not true the message defined in a command that gives an error is given, 1 is added to #ERRORCNT and the DOUNTIL loop executes again. When the error tally is zero the DOUNTIL loop ends and processing of the verified input is done.

```

DEFINE FIELD(#ERRORCNT) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
DEFINE FIELD(#NEWSALARY) REFFLD(#SALARY) LABEL('New Sal
DEFINE FIELD(#TOTSALARY) REFFLD(#SALARY) DESC('Total Salar
DEFINE FIELD(#BUDGET) REFFLD(#SALARY) LABEL('Budget') DES
GROUP_BY NAME(#XG_DTails) FIELDS(#DEPARTMENT #EMPNO #SU
DEF_LIST NAME(#EMPBROWSE) FIELDS(#XG_DTails)

```

```

BEGIN_LOOP
DOUNTIL COND(*NOERRORS)
REQUEST FIELDS(#XG_DTails #BUDGET) BROWSELIST(#EMPBRC
EXECUTE SUBROUTINE(VALIDATE)

```

```
ENDUNTIL
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
CHANGE FIELD(#TOTSALARY) TO(*DEFAULT)
SELECT FIELDS(#SALARY) FROM_FILE(PSLMST1) WITH_KEY(#DE
CHANGE FIELD(#TOTSALARY) TO('#TOTSALARY + #SALARY')
ENDSELECT
```

```
BEGINCHECK KEEP_COUNT(#ERRORCNT)
CALLCHECK FIELD(#STARTDTE) BY_CALLING(WORKDAY) PROG_1
CONDCHECK FIELD(#NEWSALARY) COND('#NEWSALARY + #TOTS
DATECHECK FIELD(#STARTDTE) IN_FORMAT(*DDMMYY) BEFORE(
FILECHECK FIELD(#EMPNO) USING_FILE(PSLMST) FOUND(*ERROR
RANGECHECK FIELD(#EMPNO) RANGE((A0000 A9999)) MSGTXT('Em
VALUECHECK FIELD(#DEPARTMENT) WITH_LIST(ADM AUD FLT GAC)
ENDCHECK IF_ERROR(*NEXT)
```

```
ENDROUTINE
```

Structuring Functions to Validate Header Fields and Multiple Details Input Via a Browse List

At times there may be a need to validate multiple values for a field or fields. An example of this would be a screen that requires input of several header fields and multiple entries of a set of detail fields. Typically a function required to do this type of validation would be structured like this:

```
FUNCTION OPTIONS(*DIRECT)
DEFINE FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
GROUP_BY NAME(#HEADER) FIELDS(... <FIELDS ON HEADER> ..... )
DEF_LIST NAME(#DETAILS) FIELDS(. <FIELDS IN DETAILS> ..... )

BEGIN_LOOP
CHANGE FIELD(#HEADER #DETAILS) TO(*NULL)
INZ_LIST NAMED(#DETAILS) NUM_ENTRYS(100) WITH_MODE(*ADD)
```

```

DUNTIL COND(*NOERRORS)
REQUEST FIELDS(#HEADER) BROWSELIST(#DETAILS)
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
END_LOOP

```

```

SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
* << >>
* << VALIDATE HEADER FIELDS HERE >>
* << >>
SELECTLIST NAMED(#DETAILS) GET_ENTRYYS(*NOTNULL)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
* << >>
* << VALIDATE DETAIL FIELDS HERE >>
* << >>
ENDCHECK IF_ERROR(*NEXT)
UPD_ENTRY IN_LIST(#DETAILS) WITH_MODE(*ADD)
ENDSELECT
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE

```

Example of Validating Header Fields and Multiple Details Input Via a Browse List

This example demonstrates how to validate a screen with several header fields and multiple values of a set of detail fields that are input via a browse list.

```

DEFINE FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
GROUP_BY NAME(#HEADER) FIELDS(#EMPNO #SURNAME #GIVEN
DEF_LIST NAME(#DETAILS) FIELDS(#SKILCODE #DATEACQ)

```

```

MESSAGE MSGTXT('Input Employee details, Press Enter.')
BEGIN_LOOP
CHANGE FIELD(#HEADER #DETAILS) TO(*NULL)
INZ_LIST NAMED(#DETAILS) NUM_ENTRYYS(100) WITH_MODE(*AD
DUNTIL COND(*NOERRORS)

```

```
REQUEST  FIELDS(#HEADER) BROWSELIST(#DETAILS)
EXECUTE  SUBROUTINE(VALIDATE)
ENDUNTIL
MESSAGE  MSGTXT('Employee details have been accepted. Input next Em
END_LOOP
SUBROUTINE NAME(VALIDATE)
CHANGE   FIELD(#ERRORCNT) TO(0)

BEGINCHECK KEEP_COUNT(#ERRORCNT)
RANGECHECK FIELD(#EMPNO) RANGE((A0001 A0090))
VALUECHECK FIELD(#SURNAME) WITH_LIST(*BLANKS) IN_LIST(*E
VALUECHECK FIELD(#GIVENAME) WITH_LIST(*BLANKS) IN_LIST(*

SELECTLIST NAMED(#DETAILS) GET_ENTRY(*NOTNULL)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
FILECHECK FIELD(#SKILCODE) USING_FILE(SKLTAB) USING_KEY(
DATECHECK FIELD(#DATEACQ)
ENDCHECK IF_ERROR(*NEXT)
UPD_ENTRY IN_LIST(#DETAILS) WITH_MODE(*ADD)
ENDSELECT

ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE
```

7.5 BROWSE

The BROWSE command is used to display selected fields from selected file records at a workstation.

Optionally, the user may select one of the displayed records. If record selection is used, the BROWSE command will return the record fields to the program in much the same way as a FETCH command does.

The use of the BROWSE command in new applications is **not** recommended.

The BROWSE command was provided in an early release of LANSAs and it will always exist for that reason.

However, subsequent releases of LANSAs have included features like application templates, pop-up windows and prompt key processing that far exceed the speed and **functionality** of the BROWSE command.

The BROWSE command **cannot** be used in programs that are portable. Use the *DBOPTIMIZE facility, request multilingual support, and GUI (Graphical User Interface) enabled or Web enabled.

Portability Considerations A build warning will be generated if used in Visual LANSAs code. An error will occur at execution time. Code using this facility can be conditioned so that it is not executed in this environment.

Also See

[7.5.1 BROWSE Parameters](#)

[7.5.2 BROWSE Examples](#)

Required

```
BROWSE ----- FIELDS -----  
- field name field attributes --->  
      |         |             | |  
      |         --- 7 max ----- |  
      ----- 100 max -----  
  
>-- FROM_FILE ---- file name . *FIRST -----  
>  
      library name
```

Optional

```
>-- WHERE ----- 'condition' ----->

>-- WITH_KEY ----- key field values ----->

>-- NBR_KEYS ----- *WITHKEY -----
->
    numeric field name

>-- GENERIC ----- *NO ----->
    *YES

>-- IO_STATUS ---- *STATUS ----->
    field name

>-- IO_ERROR ----- *ABORT -----
>
    label

>-- VAL_ERROR ---- *LASTDIS -----
>
    *NEXT
    label

>-- USE_SELECT --- *YES ----->
    *NO

>-- NO_SELECT ---- *NEXT -----
>
    label

>-- ONE_FOUND ---- *DISPLAY-----
-->
    *SELECT

>-- ISSUE_MSG ---- *NO ----->
    *YES

>-- RETURN_RRN --- *NONE -----
```

-> *field name*

>-- DOWN_SEP ----- *DESIGN -----

-> *decimal value*

>-- ACROSS_SEP --- *DESIGN -----

-> *decimal value*

>-- EXIT_KEY ----- *YES -- *EXIT ----->
*NO label

>-- MENU_KEY ----- *YES -- *MENU -----
--> *NO label

>-- ADD_KEY ----- *NO --- *NEXT -----
> *YES label

>-- CHANGE_KEY --- *NO ---- *NEXT -----
---> *YES label

>-- DELETE_KEY --- *NO ---- *NEXT -----
--> *YES label

>-- USER_KEYS ---- fnc key---'description'--label--
>

| |
----- 5 maximum -----

>-- SHOW_NEXT ---- *PRO -----|
*YES
*NO

7.5.1 BROWSE Parameters

FIELDS
FROM_FILE
WHERE
WITH_KEY
IO_ERROR
IO_STATUS
ACROSS_SEP
ADD_KEY
CHANGE_KEY
DELETE_KEY
DOWN_SEP
EXIT_KEY
GENERIC
ISSUE_MSG
MENU_KEY
NBR_KEYS
NO_SELECT
ONE_FOUND
RETURN_RRN
SHOW_NEXT
USE_SELECT
USER_KEYS
VAL_ERROR

FIELDS

Specifies either the field(s) that are to be browsed from the record in the file or the name of a group that specifies the field(s) to be browsed.

FROM_FILE

Refer to [Specifying File Names in I/O Commands](#).

WHERE

Refer to [Specifying Conditions and Expressions](#).

WITH_KEY

Refer to [Specifying File Key Lists in I/O Commands](#).

NBR_KEYS

This parameter can be used in conjunction with the WITH_KEY parameter to vary the number of key fields that are actually used to browse records at **execution time**.

*WITHKEY, which is the default value, specifies that the number of key fields will always match the number specified in the WITH_KEY parameter and the value will not be changed at execution time.

If the number of key fields is to be varied at execution time specify the name of a numeric field that contains the number of keys value. The field specified must be defined in this function or in the LANSAs data dictionary and must be numeric.

At execution time the value contained in the NBR_KEYS field must not be less than zero or greater than the number of key fields specified in the WITH_KEY parameter.

Refer to the examples following for more information.

GENERIC

Specifies whether or not generic searching is required. Generic searching is different to full or partial key searching because only the non-blank or non-zero portion of the key value is used when comparing the search key with the file key.

When using generic searching on an alphanumeric field only the leftmost non-blank portion of the search field is compared with the file key (i.e: trailing blanks are ignored for comparative purposes).

When using generic searching on a numeric field only the leftmost non-zero portion of the search field is compared with the file key (i.e: trailing zeros are ignored for comparative purposes). Also, generic numeric field comparisons are done as if both the search key and the file key are positive numbers, regardless of what they actually are.

Note that these generic search rules mean that a blank alphanumeric search key or a zero (0) numeric key will match every record selected.

For example, if a file was keyed by a name field and it contained the following values:

SM

SMIT

SMITH

SMITHS

SMITHY

SMYTHE

Then the SELECT statement:

```
SELECT WITH_KEY('SM')
```

would only select the first record in the file because it is the only record that matches the full key value 'SM'.

If however, the SELECT statement was changed to:

```
SELECT WITH_KEY('SM') GENERIC(*YES)
```

then all the records in the file would be selected because only the non-blank portion of the key value specified is compared with the file key.

***NO**, which is the default value, indicates that generic searching is not required.

***YES**, indicates that generic searching is to be performed. When generic searching is used it is actually performed only on the last key that was supplied at execution time. Other (previous) keys specified in the WITH_KEY parameter must exactly match the values in the file. They are not generically compared with the data in the file.

For instance, imagine a name and address file that is keyed by state, post/zip code and name. The command:

```
SELECT WITH_KEY('NSW' 2000 'SM') NBR_KEYS(3) GENERIC(*YES)
```

would select all names in NSW, with postcode 2000 whose names start with SM. This is an example of generic searching on an alphanumeric field. Trailing blanks are ignored when comparing the search key with the data read from the file. Also note that only the last key ('SM') is actually generically compared with the data on the file. The other keys , 'NSW' and 2000 must exactly match data read from the file.

If, at execution time the command was dynamically modified to use 2 keys, like this:

```
SELECT WITH_KEY('NSW' 2000 'SM') NBR_KEYS(2) GENERIC(*YES)
```

then it would select all names in NSW with a post code that starts with 2. This is an example of generic searching on a numeric field where trailing zeroes (0's) are ignored.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code, it must be alphanumeric with a length of 2. Even if a user field is nominated the special field IO\$STS will still be updated.

For values, refer to [I/O Return Codes](#).

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

If the default value *ABORT is not used you must nominate a valid command label to which control should be passed if an I/O error occurs.

VAL_ERROR

Specifies the action to be taken if a validation error was detected by the command.

A validation error occurs when information that is to be added, updated or deleted from the file does not pass the FILE or DICTIONARY level validation checks associated with fields in the file.

If the default value *LASTDIS is used control will be passed back to the last display screen used. The field(s) that failed the associated validation checks will be displayed in reverse image and the cursor will be positioned to the first field in error on the screen.

If the default value *LASTDIS is not used you must nominate either *NEXT, indicating that control should be passed to the next command, or, a valid command label to which control should be passed.

The *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).

When using *LASTDIS the "Last Display" must be at the same level as the database command (INSERT, UPDATE, DELETE, FETCH and SELECT). If they are at different levels e.g. the database command is specified in a SUBROUTINE, but the "Last Display" is a caller routine or the mainline, the function will abort with the appropriate error message(s).

The same does NOT apply to the use of event routines and method routines in Visual LANSA. In these cases, control will be returned to the calling routine. The fields will display in error with messages returned to the first status bar encountered in the parent chain of forms, or if none exist, the first form with a status bar encountered in the execution stack (for example, a reusable part that inherits from PRIM_OBJT).

USE_SELECT

Specifies whether the browse is to allow record selection.

The default value is *YES which means that when the browse is displayed, the user will be able to nominate which record is to be selected. Once a record selection is made, all of the fields in the function will contain the associated values and the program will continue on to the next statement.

The only other allowable value is *NO which means that the browse will be displayed with no select column. Once the enter key or a function key is pressed, the program will continue to the next statement. The value of fields within the function will be unchanged. The I/O status will return an OK if there was at least one record in the browse or else NR if there were no records in the browse.

NO_SELECT

Indicates the action to be taken if no browse entry was selected.

The default *NEXT will pass control to the next executable command.

Otherwise nominate the label associated with a command to which control should be passed.

This option will be ignored if USE_SELECT(*NO) has been specified.

ONE_FOUND

Indicates what action is to be taken if the browse command finds one and only one record matching the search criteria.

The default, *DISPLAY indicates that the record should be displayed to the user.

The only other option is to specify *SELECT which will automatically select the record. This provides a very simple means of supplying generic search lists. A partial key can be supplied, and if a single hit was found then the record would be retrieved. If more than one record was found then LANSAs would prompt the user to select the desired record.

ISSUE_MSG

Specifies whether an "end of file" message is to be automatically issued or not.

The default value is *NO which indicates that no message should be issued.

The only other allowable value is *YES which indicates that a message should be automatically issued. The message will appear on line 22/24 of the next screen format presented to the user or on the job log of a batch job.

RETURN_RRN

Specifies the name of a field in which the relative record number of the record

just selected should be returned in.

Any field nominated in this parameter must be defined within the function or the LANSAs data dictionary and must be numeric.

For further information refer also to [Load Other File](#) in the *Visual LANSAs Developers Guide*.

DOWN_SEP

Specifies the spacing between rows on the browse. The value specified must be *DESIGN or a number in the range 1 to 10. DESIGN will set a default value of 1.

ACROSS_SEP

Specifies the spacing between columns on the browse. The value specified must be *DESIGN or a number in the range 1 to 10. DESIGN will set a default value of 1.

EXIT_KEY

Specifies whether the EXIT function key is to be enabled when the browse list is displayed at the workstation. In addition it also specifies what is to happen if the EXIT key is used.

*YES, which is the default value, indicates that the EXIT key should be enabled when the screen is displayed.

If *YES is used it is also possible to nominate a command label to which control should be passed when the EXIT key is used.

If no label is specified the default value of *EXIT is used which specifies that a complete exit from the LANSAs system should be performed. In SAA/CUA partitions this is referred to as a "high" exit.

*NO indicates that the EXIT function key should not be enabled when the screen is displayed.

MENU_KEY

Specifies whether the MENU function key is to be enabled when the browse list is displayed at the workstation. In addition it also specifies what is to happen if the MENU key is used.

*YES, which is the default value, indicates that the MENU key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the MENU key is used. If no label is specified the default value of *MENU is used which

specifies that the process's main menu should be re-displayed.

*NO indicates that the MENU function key should not be enabled when the screen is displayed.

ADD_KEY

Specifies whether the ADD function key is to be enabled when the browse list is displayed at the workstation. In addition it also specifies what is to happen if the ADD key is used.

*NO, which is the default value, indicates that the ADD function key should not be enabled when the screen is displayed.

*YES indicates that the ADD key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the ADD key is used. If no label is specified the default value of *NEXT is used which specifies that the next RDML command should be executed.

CHANGE_KEY

Specifies whether the CHANGE function key is to be enabled when the browse list is displayed at the workstation. In addition it also specifies what is to happen if the CHANGE key is used.

*NO, which is the default value, indicates that the CHANGE function key should not be enabled when the screen is displayed.

*YES indicates that the CHANGE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the CHANGE key is used. If no label is specified the default value of *NEXT is used which specifies that the next RDML command should be executed.

DELETE_KEY

Specifies whether the DELETE function key is to be enabled when the browse list is displayed at the workstation. In addition it also specifies what is to happen if the DELETE key is used.

*NO, which is the default value, indicates that the DELETE function key should not be enabled when the screen is displayed.

*YES indicates that the DELETE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the DELETE key is used. If no label is specified the default value of *NEXT is used which specifies that the next

RDML command should be executed.

USER_KEYS

Specifies up to 5 additional user function keys that will be enabled when the browse list is displayed at the workstation.

Any user function keys assigned must not conflict with function keys assigned to the standard LANSAs of EXIT, MENU, MESSAGES, ADD, CHANGE or DELETE when they are enabled on a command (i.e: a function key cannot be assigned to more than one function).

Additional user function keys are specified in the format:

(fnc key number 'description' label)

where:

fnc key number Is a normal function or function key number in the range 1 to 24.

'description' Is a description of the function assigned to the function key. This description will be displayed on line 23 of the screen format. Maximum length is 8 characters.

Label Is an optional label to which control should be passed if the function key is used. Special value *NEXT is assumed as a default and indicates that the next command (after this one) should receive control.

Refer to the IF_KEY command for details of how the function key that was used can be tested in the RDML program.

As an example of use consider the following:

```
BROWSE ..... USER_KEYS((14 'Commit')(15 'Purge'))
```

```
IF_KEY WAS(*USERKEY1)
```

```
<< Commit logic >>
```

```
ENDIF
```

```
IF_KEY WAS(*USERKEY2)
```

```
<< Purge logic >>
```

ENDIF

Note that the IF_KEY command refers to the keys by symbolic names that indicate the order in which they are declared in the USER_KEYS parameter, not the actual function key numbers assigned to them. This makes changing function key assignments easier.

SHOW_NEXT

Specifies whether the "next function" field should be shown on line 22 of the screen. The next function field is a facility that allows transfer between the functions in a process without the need to return to the process menu each time. Refer to [The Function Control Table](#) in the *LANSA for i User Guide* for details about "next function" processing.

*PRO, which is the default value, indicates that the "next function" field should appear only when the process to which this function belongs has a menu selection style of "FUNCTION". If the process menu selection style is "NUMBER" or "CURSOR" then the next function field should not appear.

*YES indicates that the next function field should appear regardless of what menu selection style is being used by the process to which this function belongs.

*NO indicates that the next function field should not appear regardless of what menu selection style is being used by the process to which this function belongs.

Note: The SHOW_NEXT parameter is ignored in SAA/CUA applications.

7.5.2 BROWSE Examples

Example 1: Browse all lines of a given order allowing selection of an individual order line:

```
GROUP_BY  NAME(#BRWLIN) FIELDS(#ORDLIN #PRODUCT #QUAN  
REQUEST  FIELDS(#ORDNUM)  
BROWSE    FIELDS(#BRWLIN) FROM_FILE(ORDLIN) WITH_KEY(#
```

This is functionally similar to:

```
DEF_LIST  NAME(#BRWLIN) FIELDS((#SELECTOR *SELECT) #ORDLI  
  
REQUEST  FIELDS(#ORDNUM)  
CLR_LIST NAMED(#BRWLIN)  
SELECT   FIELDS(#BRWLIN) FROM_FILE(ORDLIN) WITH_KEY(#ORI  
ADD_ENTRY TO_LIST(#BRWLIN)  
ENDSELECT  
DISPLAY  BROWSELIST(#BRWLIN)  
SELECTLIST NAMED(#BRWLIN) GET_ENTRYS(*SELECT)  
FETCH    FIELDS(#BRWLIN) FROM_FILE(ORDLIN) WITH_RRN(#RRN  
ENDSELECT
```

Example 2: Generically browse customers by name that are in state NSW with a credit limit of \$100,000:

```
BROWSE    FIELDS(#NAME #ADD1 #ADD2 #POSTCD #CRDLIM) FROM
```

Example 3: Expand on example 2 by automatically selecting the record if it is the only record in the browse:

```
BROWSE    FIELDS(#NAME #ADD1 #ADD2 #POSTCD #CRDLIM) FROM
```

Example 4: Use browse to provide for a customer master file inquiry with update and delete facilities. The browse is to support generic searching for the customer name:

```
GROUP_BY  NAME(#BRWLIN) FIELDS(#NAME #ADD1 #ADD2 #AD  
  
BEGIN_LOOP  
L1: REQUEST FIELDS(#NAME)
```

```
SET_MODE TO(*DISPLAY)
BROWSE  FIELDS(#BRWLIN) FROM_FILE(CUSTMAS) WITH_KEY(
DISPLAY FIELDS(#BRWLIN)
IF_MODE IS(*CHANGE)
UPDATE  FIELDS(#BRWLIN) IN_FILE(CUSTMAS) WITH_RRN(#FIL
ENDIF
IF_MODE IS(*DELETE)
DELETE  FROM_FILE(CUSTMAS) WITH_RRN(#FILRRN)
ENDIF
END_LOOP
```

7.6 CALL

The CALL command is used to invoke a 3GL program or process or function and optionally pass parameters, data structures and lists into it. The CALL command can also be used in WAM Components to invoke other WEBROUTINEs in the same WAM Component or other WAM Components.

Portability Considerations See Parameters [PARM](#) and [PGM](#) and [Specifying File Names in I/O Commands](#).

Also See

[7.6.1 CALL Parameters](#)

[7.6.2 CALL Comments / Warnings](#)

[7.6.3 CALL Examples](#)

Optional

```
CALL ----- PGM ----- *NONE ----->
    pgm name
    pgm name . *LIBL
    pgm name . library name

>-- PROCESS ----- *NONE ----->
    *DIRECT
    process name

>-- FUNCTION ----- *MENU ----->
    function name

>-- WEBROUTINE --- webroutine name -----
>
    *SERVICE service name
    *EVALUATE field name

>-- ONENTRY ----- *MAP_NONE -----
>
    *MAP_ALL
    *MAP_LOCAL
    *MAP_SHARED
```

```

>-- ONEXIT ----- *MAP_NONE ----->
    *MAP_ALL
    *MAP_LOCAL
    *MAP_SHARED

>-- PARM ----- list of parameters ----->
    | expandable group expression |
    ----- 20 maximum -----

>-- EXIT_USED ---- *EXIT ----->
    *MENU
    *NEXT
    *RETURN
    label

>-- MENU_USED ---- *MENU -----
>
    *EXIT
    *NEXT
    *RETURN
    label

>-- NUM_LEN ----- *ALL15 ----->
    *DEFINED

>-- PGM_EXCH ----- *NO ----->
    *YES

>-- IF_ERROR ----- *ABORT ----->
    *NEXT
    *RETURN
    label

>-- PASS_DS ----- data structure names ----->
    |           |
    ----- 20 max -----

>-- PASS_LST ----- working list names -----|

```

| |
----- *20 max* -----

7.6.1 CALL Parameters

EXIT_USED

FUNCTION

IF_ERROR

MENU_USED

NUM_LEN

ONENTRY

ONEXIT

PARM

PASS_DS

PASS_LST

PGM

PGM_EXCH

PROCESS

WEBROUTINE

PGM

Specifies the name of a 3GL program which is to be invoked. This parameter is a qualified name. Either a program name or a process name (but not both) must be specified on this command. If required the library in which the program resides can also be specified. If no library name is specified, library *LIBL is assumed which indicates the execution time library list of the job should be searched to find the program.

The use of library names in a CALL command is **not** recommended. For further information, refer to [Specifying File Names in I/O Commands](#).

Portability Calls of 3GL programs are only supported in RDMLX programs on IBM i for compatibility with existing RDML code. As such, only RDML fields and lists are supported in the parameters PARM, PGM_EXCH, PASS_DS and PASS_LIST that may be used for 3GL program calls.

Considerations

A build warning will be generated if used in Visual LANSA code. An error will occur at execution time. Code using this facility can be conditioned so that it is not executed in this environment.

For further information refer to [Calling 3GL Programs](#) in the *LANSA Application Design Guide*.

PROCESS

Specifies the name of the LANSAs process which is to be invoked. Either a 3GL program name or a process name (but not both) must be specified.

When calling a **function** there are **large** performance benefits to be gained from using a "direct" call.

To use a direct call simply specify the name *DIRECT in this parameter, in place of the actual process name, and then nominate the function name in the FUNCTION parameter.

Note:

- A *DIRECT call does not perform security checking in the same way that a process controlled call does. This is one of the major reasons for its superior performance. Read the comments section before using the *DIRECT option.
- On IBM i, *DIRECT calls must be used when calling between RDML and RDMLX functions.
- Full RDMLX Forms cannot call a Process. Only a Function can be called.
- Forms that are NOT enabled for full RDMLX can call a Process.

FUNCTION

Optionally specifies the function within the nominated process that should be invoked. If this parameter is not specified a default value of *MENU is assumed that indicates that the main menu of the nominated process should be displayed for the user to select the desired function.

PARM

Is optional and if specified will define a list of parameters which are to be passed to the called program. The parameters must correspond to the expected parameters in the called program. This is NOT checked by LANSAs. For further information, refer to [Quotes and Quoted Strings](#). This parameter allows expandable group expressions.

Portability Considerations Not supported in the current release of Visual LANSAs and not expected to be supported in future releases.

EXIT_USED

Is valid when calling another process only. This parameter is ignored when

calling another program. Specifies what is to happen when the called process is ended by use of the EXIT function key or the EXIT command. The EXIT_USED parameter will only take effect if the EXIT function key or the EXIT command were used to end the called process.

*EXIT indicates that this function should itself terminate and request an exit from the entire LANSAs system.

*MENU indicates that this function itself should terminate and request that the process main menu be re-displayed.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

WEBROUTINE

Specifies the name of the WEBROUTINE to call. You can specify another WAM, in this case WAM name followed by a WEBROUTINE name separated by a dot (for example #MyWAM.MyWebRtn).

A Service Name can also be specified, if prefixed with *SERVICE modifier. The value can also be provided from a field, if prefixed with *EVALUATE modifier.

ONENTRY

Is valid when calling another WEBROUTINE only. For WEBROUTINE information, refer to [WEBROUTINE](#).

Used for mapping incoming fields and list into the target WEBROUTINE.

Can be one of:

*MAP_NONE does not map any fields or lists).

*MAP_ALL maps all required fields and lists.

*MAP_LOCAL only fields and lists on WEBROUTINE's WEB_MAPs are mapped.

*MAP_SHARED only WAM level WEB_MAP fields and lists are mapped, not WEBROUTINE level.

The default value is *MAP_ALL.

ONEXIT

Is valid when calling another WEBROUTINE only.

Used for mapping outgoing fields from the target WEBROUTINE.

Can be one of:

*MAP_NONE does not map any fields or lists.

*MAP_ALL maps all required fields and lists.

*MAP_LOCAL only fields and lists on WEBROUTINE's WEB_MAPs are mapped.

*MAP_SHARED only WAM level WEB_MAP fields and lists are mapped, not WEBROUTINE level.

The default value is *MAP_ALL.

MENU_USED

Is valid when calling another process only. This parameter is ignored when calling another program. Specifies what is to happen when the called process is ended by use of the MENU function key or the MENU command. The MENU_USED parameter will only take effect if the MENU/CANCEL function key or the MENU command were used to end the called process.

*EXIT indicates that this function should itself terminate and request an exit from the entire LANSAs system.

*MENU indicates that this function itself should terminate and request that the process main menu be re-displayed.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

NUM_LEN

Specifies the length to be used when passing numeric fields or literals as parameters when calling a user program.

*ALL15 is the default and indicates that all numeric parameters should be passed with length packed 15,N (where N = number of decimal places defined as per the data dictionary or DEFINE command).

*DEFINED indicates that all numeric parameters should be passed with length as per the data dictionary or DEFINE command.

PGM_EXCH

Specifies whether or not the program specified by the PGM parameter will require access to the LANSA exchange list that is normally only used to communicate information between LANSA processes and functions.

This parameter has no meaning when placing a call to a LANSA process or function (i.e: PROCESS or FUNCTION parameters used). In this context it is totally ignored, no matter what value it has.

*NO, which is the default value, specifies that the program nominated in the PGM parameter, or programs that it in turn may call, do not require access to the LANSA exchange list.

*YES specifies that the program nominated in the PGM parameter, or programs that it in turn may call, do require access to the LANSA exchange list by placing calls to program M@EXCHL.

The use of program M@EXCHL is described in detail in the section that describes the EXCHANGE command and exchange list processing.

IF_ERROR

Is valid when calling a program or a process/function.

This parameter specifies what is to happen when the called program or process/function terminates with an error.

*ABORT, which is the default value, indicates that this function itself should terminate with an error.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

PASS_DS

Allows up to 20 different data structures to be passed from an RDML function into another RDML function **or** 3GL program. The following points should be noted before attempting to use this parameter:

Each data structure name should be the name of a physical file which has been defined to LANSA and made operational.

CALL PROCESS(*DIRECT) must be specified when this parameter is used

(and the call is being made to a LANSAs function).

To be able to pass fields within the named physical file, the fields must be **referenced** at some point within the function, otherwise they will not be passed to the caller. This applies to the function being called also. Only real fields in the file can be passed, not virtual fields.

This point needs to be well understood. Say a file definition contained 3 numeric real fields called A, B and C.

An RDML function is written and it sets fields A and C to zero, then calls another RDML function that receives the structure.

If the called function started with the command "IF (B = 17)" it would **fail** because field B does not contain valid numeric data. It does not contain valid numeric data because the first function made no "reference to" or "mention of" field B at all.

The "reference" does not have to be a specific "CHANGE FIELD(B) TO(0)" type of command, it can be any reference at all in any command. It just needs to tell LANSAs that it is "interested" in field B, and therefore needs to map the current value of field B into and out of the contiguous data structure storage area before and after the call to the other function.

It is also important to note that the order in which the data structures are specified on the PASS_DS parameter of the calling function and the order in which they are specified on the RCV_DS parameter of the called function (see FUNCTION command) is significant - the data structures must appear in the same order in the called and calling functions, otherwise errors may occur.

Avoid using this facility in conjunction with the EXCHANGE command data transfer facility, particularly where a field is in both the exchange list and the data structure.

If a 3GL program is being called the data structures should be defined as received parameters in the same order as they are defined in the RDML CALL command.

A 3GL call can involve: parameters (PARM), data structures (PASS_DS) and working lists (PASS_LST). Parameters are always passed first, then all data structures, then all working lists.

The use of this facility with 3GL programs is **not** recommended unless the developer has **extensive** 3GL experience.

PASS_LST

Allows up to 20 different working list names to be passed into an RDML

function **or** 3GL program. The following points should be noted before using this parameter:

Each working list specified must be a defined working list within the function. CALL PROCESS(*DIRECT) must be specified when this parameter is used (and the call is to an RDML function).

The working lists must have been defined with the same attributes in both the calling and called function/program otherwise errors could occur.

It is also important to note that the order in which the working lists are specified on the PASS_LST parameter on the CALL command and the order in which they are specified on the RCV_LIST of the called function is significant - the working lists must appear in the same order in the called and calling functions, otherwise errors will occur.

If a 3GL program is being called the receiving data structures should be defined as received parameters in the same order as they are defined in the RDML CALL command.

A 3GL call can involve: parameters (PARM), data structures (PASS_DS) and working lists (PASS_LST). Parameters are always passed first, then all data structures, then all working lists.

Under IBM i each working list is actually passed as three 3GL level parameters:

1. Under IBM i this is the storage area representing the working list at maximum size.
2. A numeric (IBM i packed 7,0) value representing the count of the number of list entries that exist within the list area.
3. A numeric (IBM i packed 7,0) value representing the "current" list entry being processed by the caller.

The use of this facility with 3GL programs is **not** recommended unless the developer has **very extensive** 3GL experience.

7.6.2 CALL Comments / Warnings

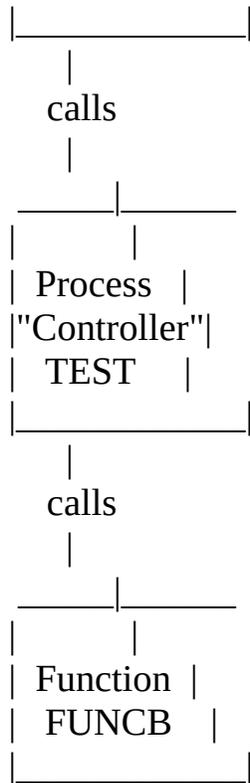
Calling a Process or Function

- The use of parameters when calling another process or function is strongly **not** recommended. Use the exchange list instead.
- When calling **another LANSAs process** the parameters must exactly match those required by the process, in number passed, and type (i.e: numeric or alpha).
- The parameters passed to the called process or function can be a field name, an alphanumeric literal, a numeric literal, a system variable or a process parameter.
- Note that when numeric parameters are defined for a process they are **always packed decimal**. Thus any numeric parameters passed to LANSAs processes should also be packed decimal with the same length and number of decimal positions. Failure to observe this rule may result in unpredictable results.
- Using multiple data structures that have fields with the same name in the same function, may cause unpredictable results.
- In the current release of LANSAs parameters passed to another process or function are **NOT RETURNED** from the process or function. However, information can be passed to and returned from a called function by using the **EXCHANGE** command just before the **CALL** command. Refer to the **EXCHANGE** command for more details of how this is done.
- Each physical file referred to by the data structures named in the **PASS_DS** parameter must be made operational or the results will be unpredictable.
- When a compiled RDML function (e.g.: **FUNCA**) is running and executes a **CALL** command like this:

```
CALL PROCESS(TEST) FUNCTION(FUNCB)
```

to invoke another RDML function called **FUNCB**, what actually happens is this:

```
_____
|      |
| Function |
| FUNCA  |
```

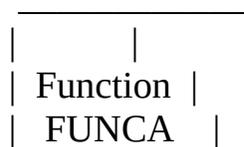


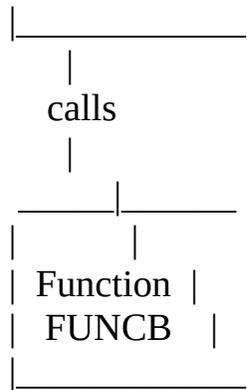
- This CALL operation can be made significantly quicker by doing the following, especially if the process controller TEST is running in interpretive mode rather than compiled mode:
 1. Include the command FUNCTION OPTIONS(*DIRECT) into FUNCB (note that is **FUNCB**) then recompile it. This simply specifies that FUNCB is eligible for direct mode invocation and it will not affect FUNCB in any way.
 2. Change the call command in **FUNCA** to be like this:

CALL PROCESS(*DIRECT) FUNCTION(FUNCB)

then recompile FUNCA. This indicates that function FUNCB should be called in direct mode, rather than via its process controller named TEST.

After these changes, the CALL will work like this:





- Use this option (i.e: FUNCTION OPTIONS(*DIRECT)) in any function that is intended to be used as a "subroutine", rather than a main controlling function directly accessible from a process menu. All calls to the function should use the CALL PROCESS(*DIRECT) option.
- This performance benefit is also available when FUNCB is used as a prompter function (i.e: it services F4=Prompt key requests). When associating FUNCB with a field in the data dictionary, simply indicate its process name as *DIRECT, rather than by specifying its actual process name (TEST).
- There is no harm in specifying FUNCTION OPTIONS(*DIRECT) in all functions. Functions using this option are equivalent to functions that do not use this option, and no restrictions on accessing such functions from process menus exist.
- In a function that does not use FUNCTION OPTIONS(*DIRECT) the associated RPG program, display file object and multilingual extension program have names F@innnnn, @innnnn and F@innnnnML respectively, where "innnnn" is an internal function identifier assigned by LANSA. This caters for duplicate function names.
- In a function that does use FUNCTION OPTIONS(*DIRECT), the resulting object names are @ffffff, @ffffff and @ffffffML, where fffffff is the name of the function (from 1 to 7 characters long). This is why function names **must** be unique when using FUNCTION OPTIONS(*DIRECT).
- Some **restrictions** do exist when using the *DIRECT facility:
 - All exchange of information between FUNCA and FUNCB must be via the exchange list. Parameters are not supported and cannot be used. This is checked by the full function checker.
 - The function name FUNCB must be unique within the partition. This

is also checked by the full function checker.

- FUNCB cannot use any "sideways" control commands like TRANSFER because in this structure it does not have its process controller (TEST) available to handle the request for it. It should **always** be terminated by a CANCEL/MENU function key or command, an EXIT function key or command, or by a RETURN command.
- FUNCB should not be part of an action bar process. This type of call, into an action bar function, would be quite strange and is unlikely to be encountered. Whenever an action bar function calls another program, the called program should only communicate with the user via POP_UPs, not via full panel DISPLAY or REQUEST commands that may upset the action bar display and processing.
- FUNCB acts like a logical "subroutine" of FUNCA.
- FUNCB should have a specific OPTIONS(*HEAVYUSAGE) or (*LIGHTUSAGE) directive. If this is not specified they will adopt the usage option of the function that called them, rather than adopting the usage option of its parent process TEST.
- The introduction of the CALL PROCESS(*DIRECT) option has now made high volume calls possible, this was previously not recommended IN LANSA. When implementing high volume calls to other functions the size of the exchange list should be considered. In these circumstances a large number of fields on the exchange list may cause a performance overhead.

Investigate the option of passing data structures (CALL...PASS_DS(#dddddd)) between functions instead of using the exchange option.

Calling a User Program

- The parameters passed to the called program or process can be a field name, an alphanumeric literal, a numeric literal, a system variable or a process parameter.
- If special value *LIBL was nominated as the library containing the program then the program should be in one of the libraries in the user's library list at the time the function is executed.
- When a parameter is alphanumeric it is always passed to the user program

with the length defined in the data dictionary or defined by the DEFINE command.

- When a parameter is numeric it is passed in a packed decimal field of length determined by the NUM_LEN parameter. If the default of *ALL15 is specified then all numeric parameters will be passed in a packed decimal 15 digit field. The number of decimals passed will match the respective definition of the field for fields and system variables. For numeric literals the number of decimals will match the number specified in the literal (i.e. 12.34 will be passed as packed (15,2); 132 will be passed as packed (15,0)).
- If *DEFINED is specified then all numeric parameters will be passed in a packed decimal field with length and number of decimals as defined in the data dictionary or defined by the DEFINE command.
- The parameters returned by the user program will only be mapped back into the function if the parameter is a field. This prevents the user program from modifying the contents of system variables, alphanumeric literals, numeric literals or process parameters.

7.6.3 CALL Examples

Example 1: Call a program named INVOICE in library PRODLIB and pass two parameters, invoice number and inquiry date as literals.

```
CALL PGM(PRODLIB/INVOICE) PARM('INV123' '010187')
```

Example 2: Call program PUTBATCH passing the fields #BATCH, #ORDER and the current date (which is obtained from system variable *DATE).

```
CALL PGM(PUTBATCH) PARM(#BATCH #ORDER *DATE)
```

Example 3: Call LANSAPROCESS ORDERS and request that the process main menu be displayed to allow the user to choose the desired function. If the user uses the MENU key to exit from process ORDERS continue processing in this function at the next RDML command:

```
CALL PROCESS(ORDERS) FUNCTION(*MENU) MENU_USED(*NEXT)
```

Example 4: Call LANSAPROCESS ORDERS and request that the function HEADER be directly invoked without displaying the process main menu. If the user uses the MENU or EXIT keys to exit from function HEADER, continue processing in this function at the next RDML command:

```
CALL PROCESS(ORDERS) FUNCTION(HEADER)
```

```
EXIT_USED(*NEXT) MENU_USED(*NEXT)
```

Example 5: Do the same thing as example 4, but use a direct call to function HEADER:

```
CALL PROCESS(*DIRECT) FUNCTION(HEADER)
```

EXIT_USED(*NEXT) MENU_USED(*NEXT)

Example 6: Pass data structure DATAFILE into function(ORDER):

This requires a file called DATAFILE to be defined to LANSAS. The real fields on this file can be passed as a data structure to another function. A dummy field on the end of the data structure would avoid the need to recompile each called function receiving the data structure each time fields are added within the data structure. As long as the length of the data structure remains consistent, the called function would not need to be recompiled (unless the existing fields are changed in length, position or type within the data structure).

CALL PROCESS(*DIRECT) FUNCTION(ORDER) PASS_DS(DATAFILE)

Example 7: Pass working list #ORDLINE to function(ORDER):

```
CALL PROCESS(*DIRECT) FUNCTION(ORDER) PASS_LST(#ORDLINE
DEF_LIST NAME(#ORDLINE) FIELDS(#ORDLINE #PRODUCT #QUANT
TYPE(*WORKING)
```

The contents of working list #ORDLINE can now be referenced by function ORDER.

Example 8: Call WEBROUTINE ORDER:

```
CALL WEBROUTINE(ORDER)
```

Values of any fields and lists specified FOR(*INPUT) on the ORDER WEBROUTINE will be passed to it.

Example 9: Call WEBROUTINE ORDER in ORDERS WAM:

```
CALL WEBROUTINE(#ORDERS.ORDER)
```

Values of any fields and lists specified FOR(*INPUT) on the ORDER WEBROUTINE will be passed to it.

Example 10: Provide the name of a WEBROUTINE to call from a field:

```
#WEBRTN := 'ORDERS.ORDER'
CALL WEBROUTINE(*EVALUATE #WEBRTN)
```


7.7 CALLCHECK

The CALLCHECK command is used to validate data by calling a program and optionally pass additional parameters to it as well as to then take appropriate action as determined by the return codes.

Portability Considerations See Parameter [PROG_TYPE](#).

Also See

[7.7.1 CALLCHECK Parameters](#)

[7.7.2 CALLCHECK Comments / Warnings](#)

[7.7.3 CALLCHECK Examples](#)

[7.4 BEGINCHECK](#)

[7.14 CONDCHECK](#)

[7.16 DATECHECK](#)

[7.35 ENDCHECK](#)

[7.48 FILECHECK](#)

[7.78 RANGECHECK](#)

[7.99 VALUECHECK](#)

Required

CALLCHECK ---- *FIELD* ----- *field name* -----
---->

>-- *BY_CALLING* --- *program name . *LIBL* -----
--->

library name

>-- *PROG_TYPE* ---- *program type* -----
>

Optional

>-- *ADD_PARM* ----- *additional parameters* -----
->

| *expandable group expression* |
----- 20 characters ----

>-- *GOOD_RET* ----- **NEXT* -----

>

**ERROR*
**ACCEPT*

>-- *BAD_RET* ----- **ERROR* -----

>

**NEXT*
**ACCEPT*

>-- *MSGTXT* ----- **NONE* ----->
message text

>-- *MSGID* ----- *DCU0005* ----->
message identifier

>

>-- *MSGF* ----- *DC@M01 . *LIBL* -----
message file . library name

>-- *MSGDTA* ----- *substitution variables* -----|
| *expandable group expression* |
----- 20 max -----

7.7.1 CALLCHECK Parameters

ADD_PARMs

BAD_RET

BY_CALLING

GOOD_RET

MSGDTA

MSGF

MSGID

MSGTXT

PROG_TYPE

FIELD

Specifies the name of the field which is to be validated.

BY_CALLING

Specifies the name of the program which is to be called to perform the validation of the field entered in the FIELD parameter. This parameter is a qualified name. The program name must be specified.

If required the library in which the program resides can also be specified. If no library name is specified, special value *LIBL is assumed which indicates the execution time library list of the function should be searched to locate the program.

PROG_TYPE

Specifies if the program to be called is a 3GL program or a function. If the program type is specified as a function, no additional parameters are allowed and the function must use option *DIRECT. Please, refer to the FUNCTION command for more details.

Portability Considerations Calling of 3GL programs is not supported in the current release of Visual LANSA but will be supported in a future release.

A build warning will be generated if used in Visual LANSA code. An error will occur at execution time. Code using this facility can be conditioned so that it is not executed in this environment.

For further information refer to [Calling 3GL Programs](#) in the *LANSA Application Design Guide*.

ADD_PARMS

Specifies any additional parameters which should be passed to the program. See comments following for more details.

GOOD_RET

Specifies the action to be taken if the user program nominated in the BY_CALLING parameter gives a "good" return code. Refer to the comments section for more details of return codes.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the return code will be accepted and no further validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

BAD_RET

Specifies the action to be taken if the user program nominated in the BY_CALLING parameter gives a "bad" return code. Refer to the comments section for more details of return codes.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues

with the next RDML command.

If *ACCEPT is specified the return code will be accepted and no further validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

"&1 are out of stock ... reorder &2"

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this:

```
MSGDTA('BOLTS   ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA('"BOLTS   "' #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.7.2 CALLCHECK Comments / Warnings

- The CALLCHECK command must be coded within a BEGINCHECK / ENDCHECK validation block. Refer to these commands for further details.
- All programs called as part of a complex logic check must have at least 3 standard parameters. These parameters are **implicit** and do not have to be declared in the ADD_PARMS parameter.

Name Description

Return code Alphanumeric length 1. Returned by the program as '1' (good return) or '0' (bad return). Used by the program to indicate to LANSAs the success or failure of the complex logic check.

Name of field Alphanumeric length 10. Passed to the program. Contains the name (as opposed to the value) of the field that is passed in the third parameter.

Value of field Length and type depend upon the data dictionary definition of the field. Alphanumeric fields are passed with same type and length as their data dictionary definition. All numeric fields (type P or S) are passed as packed (type P) and the same length and number of decimal positions as their data dictionary definition. Note that the value of the field is passed in a work area, thus it is not possible to change value of the field by changing the value of the parameter in the validation program.

- Additional parameters may be passed to the program by nominating them in the ADD_PARMS parameter.
- Additional parameters may be:
 - An alphanumeric literal
 - A numeric literal such as 1, 14.23, -1.141217.
 - Another field name such as #CUSTNO, #INVNUM, etc.
 - A system variable name such as *BLANKS, *ZERO, *DATE or any other system variables defined at your installation.
 - A process parameter such as *UP01, *UP02, etc.
 - an expandable group expression such as (#XG_CUST #XG_PROD

*EXCLUDING #CUSTNO).

- The type and length of the parameter(s) passed depends upon the type and length of the parameter value supplied.
- For alphanumeric fields (alpha literals, alpha fields, alpha system variables or alpha process parameters) the parameter is passed as alpha (256) with the parameter value left aligned into the 256 byte parameter.
- For numeric fields (numeric literals, numeric fields, numeric system variables or numeric process parameters) the parameter is passed as packed 15 with the same number of decimal positions as the parameter value.
- For numeric literals this means the same number of decimal positions as specified in the literal (e.g.: 1.12 will be passed as packed 15,2. 7.12345 will be passed as packed 15,5. 143 will be passed packed 15,0. etc).
- For all other types of numeric parameters this means the same number of decimal positions as their respective definitions.
- As with the standard parameters, the actual value is passed in a work area so it is not possible to change the value of a field by changing the parameter value in the validation program.
- Refer to the [Complex Logic Rule](#) in the *LANSAP for i User Guide* for further information.

7.7.3 CALLCHECK Examples

[Structuring Functions for Inline Validation](#)

[Structuring Functions to Use a Validation Subroutine](#)

[Using the CALLCHECK Command for Inline Validation](#)

[Using the CALLCHECK Command for Validation with a Subroutine](#)

Structuring Functions for Inline Validation

Typically functions using validation commands (eg: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for inline validation like this:

```
BEGIN_LOOP
REQUEST  << INPUT >>
BEGINCHECK
*        << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK
*        << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is passed back to the REQUEST command. This happens because of the default IF_ERROR(*LASTDIS) parameter on the ENDCHECK command.

Structuring Functions to Use a Validation Subroutine

Typically functions using validation commands (eg: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for subroutine validation like this:

```
DEFINE   FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')

BEGIN_LOOP
DUNTIL  COND(*NOERRORS)
REQUEST << INPUT >>
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
```

```
*      << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE   FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
*      << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is returned to the main function loop with #ERRORCNT > 0.

Using the CALLCHECK Command for Inline Validation

This example demonstrates how to use the CALLCHECK command within the main program block with an RDML function as the validation program.

After the user enters the requested details, the start date is checked to make sure it falls on a working day by calling the WORKDAY RDML program. If the WORKDAY program returns a negative response the defined message is given and program control returns to the last screen displayed, the Request screen in this case.

```
DEF_LIST  NAME(#EMPBROWSE) FIELDS(#EMPNO #STARTDTE)

BEGIN_LOOP
REQUEST  FIELDS(#EMPNO #STARTDTE) BROWSELIST(#EMPBROW

BEGINCHECK
CALLCHECK FIELD(#STARTDTE) BY_CALLING(WORKDAY) PROG_]
ENDCHECK

ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

The WORKDAY function is defined as follows;

```
FUNCTION  OPTIONS(*DIRECT *LIGHTUSAGE *MLOPTIMISE *NUM
```

```

DEFINE FIELD(#TESTDATEC) TYPE(*CHAR) LENGTH(6)
DEFINE FIELD(#TESTDATEN) LENGTH(6) DECIMALS(0) REFFLD(#I
DEFINE FIELD(#DAYOFWEEK) TYPE(*CHAR) LENGTH(3)

CHANGE FIELD(#TESTDATEN) TO(#VALFLD$NV)
USE BUILTIN(CONVERTDATE) WITH_ARGS(#TESTDATEC B R) TC
CASE OF_FIELD(#DAYOFWEEK)
WHEN VALUE_IS('= MON' '= TUE' '= WED' '= THU' '= FRI')
CHANGE FIELD(#VALFLD$RT) TO("'1'")
OTHERWISE
CHANGE FIELD(#VALFLD$RT) TO("'0'")
ENDCASE

RETURN

```

For more information related to creating complex logic validation functions see the technical notes for *ALP_FIELD_VALIDATE and *NUM_FIELD_VALIDATE in the parameters section for the FUNCTION command.

Using the CALLCHECK Command for Validation with a Subroutine

This example demonstrates how to use the CALLCHECK command inside a subroutine with an RDML function as the validation program.

After the user enters the requested input code the VALIDATE subroutine is called. It checks that the input code is in the correct format by calling the ANUMBER RDML program. If the ANUMBER program returns a negative response the message defined in the CALLCHECK command is given and the DOUNTIL loop executes again. When the ANUMBER program returns a positive response the DOUNTIL loop ends and processing of the verified input is done.

```

DEFINE FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
DEFINE FIELD(#INPUT) TYPE(*CHAR) LENGTH(7)
DEF_LIST NAME(#EMPBROWSE) FIELDS(#INPUT)

BEGIN_LOOP
DOUNTIL COND(*NOERRORS)
REQUEST FIELDS(#INPUT) BROWSELIST(#EMPBROWSE)

```

```

EXECUTE SUBROUTINE(VVALIDATE)
ENDUNTIL

ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP

SUBROUTINE NAME(VVALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)

BEGINCHECK KEEP_COUNT(#ERRORCNT)
CALLCHECK FIELD(#INPUT) BY_CALLING(ANUMBER) PROG_TYPE
ENDCHECK IF_ERROR(*NEXT)

ENDROUTINE

```

The ANUMBER program is defined as follows:

```

FUNCTION OPTIONS(*DIRECT *LIGHTUSAGE *MLOPTIMISE *ALP_
DEFINE FIELD(#INPUT) TYPE(*CHAR) LENGTH(7)
DEFINE FIELD(#INPUTA1) TYPE(*CHAR) LENGTH(1) TO_OVERLAY
DEFINE FIELD(#INPUTA6) TYPE(*CHAR) LENGTH(6) TO_OVERLAY
DEFINE FIELD(#DECIMAL) TYPE(*DEC) LENGTH(1) DECIMALS(1)
DEFINE FIELD(#RTN_CODE) TYPE(*CHAR) LENGTH(1)

CHANGE FIELD(#INPUT) TO(#VALFLD$AV)
USE BUILTIN(CHECKNUMERIC) WITH_ARGS(#INPUTA6 6 0) TO_(

IF COND('( #INPUTA1 = A) and (#RTN_CODE = Y)')
CHANGE FIELD(#VALFLD$RT) TO("1")
ELSE
CHANGE FIELD(#VALFLD$RT) TO("0")
ENDIF

RETURN

```

For more information related to creating complex logic validation functions see the technical notes for *ALP_FIELD_VALIDATE and *NUM_FIELD_VALIDATE in the parameters section for the FUNCTION command.

7.8 CASE

The CASE command is used to specify the beginning of a case condition. The CASE command is used in conjunction with the WHEN, OTHERWISE and ENDCASE commands.

Refer to the WHEN, OTHERWISE and ENDCASE commands for more details and examples of these commands.

Also See

[7.8.1 CASE Parameters](#)

[7.8.2 CASE Comments / Warnings](#)

[7.8.3 CASE Examples](#)

[7.34 ENDCASE](#)

[7.73 OTHERWISE](#)

[7.100 WHEN](#)

Required

CASE ----- OF_FIELD ----- field name -----
-|

7.8.1 CASE Parameters

OF_FIELD

Specifies the name of the field which is to be compared to the condition(s) specified in the WHEN commands that follow.

7.8.2 CASE Comments / Warnings

- Note that only the commands associated with one 'WHEN VALUE_IS' condition can be executed. If more than one 'WHEN VALUE_IS' condition is true, within a case command, only the commands associated with the first true condition are executed.

7.8.3 CASE Examples

Basic CASE Processing

Why use the CASE command?

Using the CASE command with OR operations

Using the CASE command with operations other than "=" (equal to)

Using the CASE command with compound expressions

Basic CASE Processing

This example illustrates the most basic type of CASE command processing. Imagine that a user inputs a value into field #DEPARTMENT and that a CASE command to be used to implement this logic:

When user inputs this value into field #DEPARTMENT	Field #SECTION should be changed to this value . . .	And this message should be issued . . .
ADM	A	ADM was entered
AUD	B	AUD was entered
FLT	C	FLT was entered
TRVL	D	TRVL was entered
Any other value	E	Other value was entered

Structurally, a CASE command to implement this logic would look like this:

```
CASE    OF_FIELD(#DEPARTMENT)
WHEN    VALUE_IS(= ADM')
        << Logic >>
WHEN    VALUE_IS(= AUD')
        << Logic >>
WHEN    VALUE_IS(= FLT')
        << Logic >>
WHEN    VALUE_IS(= TRVL')
```

```

    << Logic >>
OTHERWISE
    << Logic >>
ENDCASE

```

A sample program to fully implement this logic might be coded like this:

```

BEGIN_LOOP
REQUEST  FIELDS(#DEPARTMENT (#SECTION *OUTPUT))
CASE    OF_FIELD(#DEPARTMENT)
WHEN    VALUE_IS(= ADM')
CHANGE  FIELD(#SECTION) TO(A)
MESSAGE MSGTXT('ADM was entered')
WHEN    VALUE_IS(= AUD')
CHANGE  FIELD(#SECTION) TO(B)
MESSAGE MSGTXT('AUD was entered')
WHEN    VALUE_IS(= FLT')
CHANGE  FIELD(#SECTION) TO(C)
MESSAGE MSGTXT('FLT was entered')
WHEN    VALUE_IS(= TRVL')
CHANGE  FIELD(#SECTION) TO(D)
MESSAGE MSGTXT('TRVL was entered')
OTHERWISE
CHANGE  FIELD(#SECTION) TO(E)
MESSAGE MSGTXT('Other value was entered')
ENDCASE
END_LOOP

```

Why use the CASE command?

Imagine you have to implement this logic:

When field #DEPTMENT is this value ...	Field #SECTION should be changed to this value ...	And this message should be issued
ADM or FLT or TRVL	A	Either ADM , FLT or TRVL was entered
AUD or INF or MKT	B	Either AUD, INF or

		MKT was entered
Any other value	C	Other value was entered

If you code nested IF-ELSE-END blocks you will end up with logic like this:

```

IF      COND('(#DEPARTMENT = ADM) or (#DEPARTMENT = FLT) or (#DEPT
CHANGE  FIELD(#SECTION) TO(A)
MESSAGE MSGTXT('Either ADM , FLT or TRVL was entered')
ELSE
IF      COND('(#DEPARTMENT = AUD) or (#DEPARTMENT = INF) or (#DEPT
CHANGE  FIELD(#SECTION) TO(B)
MESSAGE MSGTXT('Either AUD, INF or MKT was entered')
ELSE
CHANGE  FIELD(#SECTION) TO(C)
MESSAGE MSGTXT('Other value was entered')
ENDIF
ENDIF

```

However, by using a CASE command you can code this instead:

```

CASE    OF_FIELD(#DEPARTMENT)
WHEN    VALUE_IS('= ADM' '= FLT' '= TRVL')
CHANGE  FIELD(#SECTION) TO(A)
MESSAGE MSGTXT('Either ADM, FLT or TRVL was entered')
WHEN    VALUE_IS('= AUD' '= INF' '= MKT')
CHANGE  FIELD(#SECTION) TO(B)
MESSAGE MSGTXT('Either AUD, INF or MKT was entered')
OTHERWISE
CHANGE  FIELD(#SECTION) TO(C)
MESSAGE MSGTXT('Other value was entered')
ENDCASE

```

Which is shorter, easier to read and easier to maintain. It also executes faster in most situations.

Using the CASE command with OR operations

Imagine that when field #TEST contains A or B or C you have to perform some

operation.

If it contains X or Y or Z you have to perform some other operation.

If it contains any other value it all then you have to perform yet another operation.

You could use a CASE command structured like this to implement the required logic:

```
CASE    OF_FIELD(#TEST)

WHEN    VALUE_IS('= A' '= B' '= C')

        << Logic Block 1 >>

WHEN    VALUE_IS('= X' '= Y' '= Z')

        << Logic Block 2>>

OTHERWISE

        << Logic Block 3>>

ENDCASE
```

Using the CASE command with operations other than "=" (equal to)

Imagine that when field #COPIES needs to be validated according to these rules:

Value of #COPIES	Message to Issue
= 0	Value of zero is invalid
< 0	Value is negative !
1 to 50	Value is sensible
> 50	Value is probably incorrect

Hence a CASE command can be used like this:

```
DEFINE FIELD(#COPIES) TYPE(*DEC) LENGTH(3) DECIMALS(0) ED
BEGIN_LOOP
REQUEST FIELDS(#COPIES)
CASE OF_FIELD(#COPIES)
WHEN VALUE_IS(= 0)
MESSAGE MSGTXT('Value of zero is invalid')
WHEN VALUE_IS(< 0)
MESSAGE MSGTXT('Value is negative')
WHEN VALUE_IS(<= 50)
MESSAGE MSGTXT('Value is sensible')
OTHERWISE
MESSAGE MSGTXT('Value is probably incorrect')
ENDCASE
END_LOOP
```

Using the CASE command with compound expressions

In a situation whereby a field #DISCOUNT needs to be validated by using a mathematical calculation such as this:

When user inputs this value into fields #DISCOUNT, #QUANTITY	This equation is used to validate the discount value	A message will be displayed
1000, 20	#QUANTITY * 0.1	Discount value is larger than QUANTITY times 0.1
10, 200	#QUANTITY * 0.1	Discount value is less than QUANTITY times 0.1
10, 100	#QUANTITY * 0.1	A correct discount value was entered

This can be coded by using the CASE command:

```
DEFINE FIELD(#DISCOUNT) TYPE(*DEC) LENGTH(11) DECIMALS(2)
```

```
DEFINE FIELD(#QUANTITY) TYPE(*DEC) LENGTH(3) DECIMALS(0)
BEGIN_LOOP
REQUEST FIELDS(#DISCOUNT #QUANTITY)
CASE OF_FIELD(#DISCOUNT)
WHEN VALUE_IS(> (#QUANTITY * 0.1))
MESSAGE MSGTXT('Discount value is larger than quantity times 0.1')
WHEN VALUE_IS(< (#QUANTITY * 0.1))
MESSAGE MSGTXT('Discount value is less than quantity times 0.1')
OTHERWISE
MESSAGE MSGTXT('A correct discount value was entered')
ENDCASE
END_LOOP
```

7.9 CHANGE

The CHANGE command is used to change the value(s) of a field(s).

The FIELD parameter specifies the name of the field or fields that are to have their contents changed by the command.

Optionally the FIELD parameter may specify a group name or an expandable group expression which indirectly indicates that a group or list of fields are to have their contents changed by the command.

When the FIELD parameter nominates a group or list name, rather than specific field(s), the VALUE parameter may **only** be one of the special values *NULL, *SQLNULL, *DEFAULT, *NAVAIL, *LOVAL or *HIVAL. This constraint doesn't apply when an expandable group expression is used, as the expression is replaced by the fields themselves.

The VALUE parameter allows an expression to be specified that is used to change the contents of the field(s) nominated in the FIELD parameter.

Portability Considerations Refer to Parameters: [PRECISION](#), [ROUND_UP](#) and [TO](#). Refer also to [Portability Specifics](#) in the *LANSA Application Design Guide*.

Also See

[7.9.1 CHANGE Parameters](#)

[7.9.2 CHANGE Comments/Warnings](#)

[7.9.3 CHANGE Examples](#)

Required

```
CHANGE ----- FIELD ----- field / list name(s) -----  
->  
    expandable group expression  
  
>-- TO ----- value ----->
```

Optional

```
>-- PRECISION ---- *COMPUTE ---  
- *COMPUTE ----->
```

total digits total decimals

>-- *ROUND_UP* ----- *NO -----|
 *YES

7.9.1 CHANGE Parameters

FIELD

PRECISION

ROUND_UP

TO

FIELD

Specifies the name(s) of the field(s), list(s) or expandable group expression which is/are to be changed.

TO

Specifies the new value which field(s) nominated in the FIELD parameter are to take on. The new value specified can be another field name, an alphanumeric literal, a numeric literal, a system variable, a process parameter or an expression involving any of these in combination.

The VALUE parameter may also be specified as any one of the following "special" values when using an RDML field:

- *NULL Indicates all alphanumeric fields should be set to blanks and all numeric fields to zero.
- *NAvail Indicates all numeric fields should be set to zero and all alphanumeric fields of less than 3 characters to blanks. Alphanumeric fields of 3 or more characters in length are to be set to as much of "N/AVAIL" as will fit into the field.
- *DEFAULT Indicates all fields should be set to their data dictionary or program defined default values.
- *HIVAL Indicates that all fields should be set to their highest possible value. For alphanumeric fields this means all bytes are set to hexadecimal X'FF'. For numeric fields this means all digits are set to 9 and the sign is made to be positive.
- *LOVAL Indicates that all fields should be set to their lowest possible value. For alphanumeric fields this means all bytes are set to hexadecimal X'00'. For numeric fields this means all digits are set to 9 and the sign is made to be negative.

The following special options can be used in Visual LANSA applications:

Change Command Special Option	Meaning
*REMEMBERED_VALUE_FOR_USER	Remembers the value of the specified field for the current user in the context of the current form or function.
*REMEMBERED_VALUE_FOR_USER_IN_SYSTEM	Remembers the value of the specified field for the current user in all contexts (within the current PC system).
*REMEMBERED_VALUE_FOR_FUNCTION	Remembers the value of the specified field for all users in the context of the current form or function.
*REMEMBERED_VALUE_FOR_SYSTEM	Remembers the value of the specified field for all users in all contexts (within the current PC system).

The following table describes the behavior of RDMLX fields for special values. (Note that the behaviour of RDMLX Packed and Signed fields are the same as RDML fields of the same type.)

Special Value	FIELD type	Notes
*NULL	Integer, Float, or Boolean	Treat as per Signed/Packed, i.e. *ZERO (For Boolean *ZERO means Off/False)
*NULL	Date or Datetime	1900-01-01

*NULL	Time	Midnight (00:00:00)
*NULL	Char, String, Binary, VarBinary, CLOB, BLOB	Empty string. Note that this is considered equivalent to *BLANKS for comparisons. BLOB & CLOB: Filename is empty.
*NAVAIL	Integer, Float, Time or Boolean	As for Signed/Packed, i.e. *ZERO
*NAVAIL	Date, Datetime, Binary, VarBinary	FFC Error as cannot put character string into these fields
*NAVAIL	Char, String, CLOB, BLOB	As for Alpha. BLOB & CLOB: Filename is " N/AVAILABLE " .
*DEFAULT	Any	Field default
*SQLNULL	Any	If the field does not have the ASQN attribute it is an FFC Error. Otherwise it is okay.
*HIVAL *LOVAL	Integer	Set to the maximum value (positive) for the integer. Set to the minimum value (negative) for the integer. If field has SUNS attribute, minimum is *ZERO.
*HIVAL *LOVAL	Boolean	Set to On/True. Set to Off/False.
*HIVAL *LOVAL	Float, Date, Time	FFC Error as no logical upper or lower, especially for Date, as

		different databases support different values.
*HIVAL *LOVAL	Datetime	*TIMESTAMP_HIVAL *TIMESTAMP_LOVAL
REMEMBERED_VALUE	CLOB, BLOB, Date, Time, Datetime, Integer, Float, Boolean	Store in the registry as a string, as per current fields.
REMEMBERED_VALUE	Char, String	There may be a limit to the size of the string that may be stored (Char/String allow up to 64Kb). If there is a limit, and the field size is greater than the limit, FFC Error. If there is no limit, or the field size is less than the limit, store in the registry as a String , as per current fields.
REMEMBERED_VALUE	Binary, VarBinary	There may be a limit to the size of the string that may be stored (Binary/VarBinary allow up to 32Kb). If there is a limit, and the field size is greater than the limit, FFC Error. If there is no limit, or the field size is less than the limit, store in the registry as a Binary .

PRECISION

Specifies, for numeric expressions only, what precision is to be used for any

intermediate work fields generated by the RDML compiler.

When a numeric expression is evaluated the RDML compiler, by default, attempts to compute the precision of any intermediate work fields. This is what happens when this parameter is not specified, or specified using the default values.

This logic is biased towards the precision of leading (or significant) digits. In some cases it may cause the loss of decimal precision. The PRECISION parameter is provided to allow you to manually specify the precision required in intermediate work fields.

For instance, if you multiply a 9,2 number by a 15,9 number, the logic determines that one of the numbers has 7 leading digits (the 9,2 one). Thus intermediate calculations will use a 15,8 work field which also has 7 leading digits. This may cause the loss of decimal precision.

To overcome this problem, and force intermediate work fields to use a predetermined precision, use the PRECISION parameter.

This parameter has 2 values. The first specifies the total number of digits (including decimals) that all intermediate work fields are to have. The second indicates how many of these are to be decimals. When specifying a PRECISION parameter you must specify both the total number of digits and the number of decimals.

For instance, PRECISION(15 6) indicates a total of 15 digits, 6 of which are decimals. Thus the work field has 9 leading digits and 6 decimals.

It would be unusual to code a PRECISION parameter in which the number of decimals required was less than the number of decimals in any individual field involved in the expression.

Also See

[Field Types](#)

ROUND_UP

Specifies, for numeric and datetime expressions only, whether or not the result of the expression is to be "rounded up" before it is placed into the result field nominated in the FIELD parameter.

If this parameter is not specified, or specified using value *NO, then any decimal portion is truncated when it does not fit into the result field.

For instance, if you divide two integer values containing 3 and 2 and place the result into an integer field, you will get a result of 1.

However, if you use the ROUND_UP parameter, and an appropriate PRECISION parameter, you will get a resulting integer value of 2.

Always use the PRECISION parameter in conjunction with the ROUND_UP parameter to ensure that all intermediate work fields used to evaluate the expression have **at least** one more decimal position than the field nominated in the FIELD parameter.

If the intermediate work fields involved do not have at least one more decimal position than the field nominated in the FIELD parameter, then the ROUND_UP parameter will be ignored and treated as if ROUND_UP(*NO) was specified.

Datetime fields can have between 0 - 9 fractional seconds. **The default behaviour on assignment of a long Datetime to a shorter Datetime will be to truncate fractional seconds.** If ROUND_UP is specified, rounding of fractional seconds up will occur when this is appropriate.

The use of the ROUND_UP parameter equates directly to the RPG "half adjust" facility. In fact, using the ROUND_UP parameter simply causes the final mapping of the intermediate work field into the result field (in the translated RPG) to use the RPG "half adjust" option (provided that the intermediate work field has more decimal positions than the result field).

You may choose to refer to the appropriate RPG manual for more details of the mechanics of the "half adjust" facility.

Portability Considerations When this value is specified as *YES, you must specify an appropriate PRECISION value to ensure platform consistency in the results obtained. Failure to observe this rule may lead to rounding variances when your code is ported to different platforms.

7.9.2 CHANGE Comments/Warnings

- When changing multiple variables simultaneously to an expression, the position of the field will affect the result as shown in examples A and B.

Define Field(#STD_NUMSV) Reffld(#STD_NUM)

Example A: #std_num gets value 5 and #std_numsv gets value 6

Change Field(#STD_NUM) To(4)

Change Field(#STD_NUM #STD_NUMSV) To('#std_num + 1')

Example B: #std_num gets value 5 and #std_numsv gets value 5

Change Field(#STD_NUM) To(4)

Change Field(#STD_NUMSV #STD_NUM) To('#std_num + 1')

Using quotes

- In **RDML functions**, a literal whose first character looks like a number (begins with plus, minus, decimal point or a digit), must be triple quoted using three single quotes. For example:

```
CHANGE FIELD(#ADDRESS1) TO('1 Mount ST')
```

- **In RDMLX code:**

- The literal needs to be enclosed in double quote characters:

```
CHANGE FIELD(#ADDRESS1) TO("1 Mount ST")
```

- When assigning the value, the literal can be in either single or double quotes:

```
#ADDRESS1 := '1 Mount ST'
```

or

```
#ADDRESS1 := "1 Mount ST"
```

7.9.3 CHANGE Examples

Example 1: Increment field #COUNT.

```
CHANGE FIELD(#COUNT) TO('#COUNT + 1')
```

Example 2: Add field #QUANTITY to field #COUNT.

```
CHANGE FIELD(#COUNT) TO('#COUNT + #QUANTITY')
```

Example 3: Change all fields in group #ORDERLINE to their null values. A "null" value is blanks for an alphanumeric field and zero for a numeric field.

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU
```

```
CHANGE FIELD(#ORDERLINE) TO(*NULL)
```

which is identical to:

```
CHANGE FIELD(#ORDLIN) TO(*NULL)
CHANGE FIELD(#PRODUCT) TO(*NULL)
CHANGE FIELD(#QUANTITY) TO(*NULL)
CHANGE FIELD(#PRICE) TO(*NULL)
```

which is identical to:

```
CHANGE FIELD(#ORDLIN) TO(*ZERO)
CHANGE FIELD(#PRODUCT) TO(*BLANKS)
CHANGE FIELD(#QUANTITY) TO(*ZERO)
CHANGE FIELD(#PRICE) TO(*ZERO)
```

which is identical to:

```
CHANGE FIELD(#ORDLIN) TO(0)
CHANGE FIELD(#PRODUCT) TO('' '')
CHANGE FIELD(#QUANTITY) TO(0)
CHANGE FIELD(#PRICE) TO(0)
```

Example 4: Change all fields in group #ORDERLINE to their data dictionary defined default values.

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU
```

```
CHANGE FIELD(#ORDERLINE) TO(*DEFAULT)
```

which is identical to:

```
CHANGE FIELD(#ORDLIN) TO(*DEFAULT)
CHANGE FIELD(#PRODUCT) TO(*DEFAULT)
CHANGE FIELD(#QUANTITY) TO(*DEFAULT)
CHANGE FIELD(#PRICE) TO(*DEFAULT)
```

Example 5: Change fields #A, #B and #C so that they all contain the product of #QUANTITY and #PRICE.

```
CHANGE FIELD(#A #B #C) TO('#QUANTITY * #PRICE')
```

which is identical to:

```
CHANGE FIELD(#A) TO('#QUANTITY * #PRICE')
CHANGE FIELD(#B) TO('#QUANTITY * #PRICE')
CHANGE FIELD(#C) TO('#QUANTITY * #PRICE')
```

Example 6: Change fields #A, #B and #C so that they all contain the value 1.

```
CHANGE FIELD(#A #B #C) TO(1)
```

which is identical to:

```
CHANGE FIELD(#A) TO(1)
CHANGE FIELD(#B) TO(1)
CHANGE FIELD(#C) TO(1)
```

Example 7: Change fields #A and #B so that they contain the product of '#A + 1'.

```
CHANGE FIELD(#A #B) TO('#A + 1')
```

which is identical to:

```
CHANGE FIELD(#A) TO('#A + 1')
CHANGE FIELD(#B) TO('#A + 1')
```

Note that this shows that the resulting values of #A and #B are not the same. The value of #B in this case is 1 more than #A.

This is not necessarily wrong, but it may not be the expected result. If you wanted both #A and #B to have the same value ($\#A + 1$), then one of the following methods should be used.

```
CHANGE FIELD(#B #A) TO(#A + 1)
```

which is identical to:

```
CHANGE FIELD(#B) TO('#A + 1')  
CHANGE FIELD(#A) TO('#A + 1')
```

or

```
CHANGE FIELD(#A) TO('#A + 1')  
CHANGE FIELD(#B) TO(#A)
```

7.10 CHECK_FOR

The CHECK_FOR command is used to check for the existence of a record in a particular file.

Portability Considerations Refer to parameter [IN_FILE](#).

Also See

[7.10.1 CHECK_FOR Parameters](#)

[7.10.2 CHECK_FOR Comments / Warnings](#)

[7.10.3 CHECK_FOR Examples](#)

Required

```
CHECK_FOR ---- INFILE ----- file name . *FIRST -----  
----->
```

library name

```
>-- WITH_KEY ----- key field values ----->  
      expandable group expression
```

Optional

```
>-- IO_STATUS ---- *STATUS ----->  
      field name
```

```
>-- IO_ERROR ----- *ABORT -----  
>  
      *NEXT  
      *RETURN  
      label
```

```
>-- VAL_ERROR ---- *LASTDIS -----  
>  
      *NEXT  
      *RETURN  
      label
```

```
-> >-- NOT_FOUND ---- *NEXT -----  
      *RETURN  
      label  
  
>-- ISSUE_MSG ---- *NO -----|  
      *YES
```

7.10.1 CHECK_FOR Parameters

[IN_FILE](#)

[IO_ERROR](#)

[IO_STATUS](#)

[ISSUE_MSG](#)

[NOT_FOUND](#)

[VAL_ERROR](#)

[WITH_KEY](#)

IN_FILE

Refer to [Specifying File Names in I/O Commands](#).

WITH_KEY

Refer to [Specifying File Key Lists in I/O Commands](#).

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS will still be updated.

For the values, refer to [I/O Return Codes](#).

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command. The purpose of *NEXT is to permit you to handle error messages in the RDML, and

then ABORT, rather than use the default ABORT. (It is possible for processing to continue for LANSA for i and Visual LANSA, but this is NOT a recommended way to use LANSA.)

ER returned from a database operation is a fatal error and LANSA does not expect processing to continue. The IO Module is reset and further IO will be as if no previous IO on that file had occurred. Thus you must not make any presumptions as to the state of the file. For example, the last record read will not be set. A special case of an IO_ERROR is when a trigger function is coded to return ER in TRIG_RETC. The above description applies to this case as well. Therefore, LANSA recommends that you do NOT use a return code of ER from a trigger function to cause anything but an ABORT or EXIT to occur before any further IO is performed.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

VAL_ERROR

Specifies the action to be taken if a validation error was detected by the command.

A validation error occurs when information that is to be added, updated or deleted from the file does not pass the FILE or DICTIONARY level validation checks associated with fields in the file.

If the default value *LASTDIS is used control will be passed back to the last display screen used. The field(s) that failed the associated validation checks will be displayed in reverse image and the cursor positioned to the first field in error on the screen.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

<p>The *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).</p>

When using *LASTDIS the "Last Display" must be at the same level as the database command (INSERT, UPDATE, DELETE, FETCH and SELECT). If they are at different levels e.g. the database command is specified in a SUBROUTINE, but the "Last Display" is a caller routine or the mainline, the function will abort with the appropriate error message(s).

The same does NOT apply to the use of event routines and method routines in Visual LANSA. In these cases, control will be returned to the calling routine. The fields will display in error with messages returned to the first status bar encountered in the parent chain of forms, or if none exist, the first form with a status bar encountered in the execution stack (for example, a reusable part that inherits from PRIM_OBJT).

NOT_FOUND

Specifies what is to happen if no record is found in the file that has a key matching the key nominated in the WITH_KEY parameter.

*NEXT indicates that control should be passed to the next command.

*RETURN indicates that control should be returned to the invoking routine (identical to executing a RETURN command).

If none of the previous values are used you must nominate a valid command label to which control should be passed.

ISSUE_MSG

Specifies whether a "not found" message is to be automatically issued or not.

The default value is *NO which indicates that no message should be issued.

The only other allowable value is *YES which indicates that a message should be automatically issued. The message will appear on line 22/24 of the next screen format presented to the user or on the job log of a batch job.

7.10.2 CHECK_FOR Comments / Warnings

- The CHECK_FOR command does not read fields from the file into the RDML program. It only checks if a record with the key specified actually exists in the file specified. To read fields into the program use the FETCH command or SELECT command.

7.10.3 CHECK_FOR Examples

Example 1: Check for a record in the file CUSMST that has a key matching the value currently in the field #CUSNUM:

```
CHECK_FOR IN_FILE(CUSMST) WITH_KEY(#CUSNUM)

IF_STATUS IS_NOT(*EQUALKEY)
ABORT MSGTXT('Customer not found in customer master')
ENDIF
```

Note how the CHECK_FOR command returns a status of *EQUALKEY or *NOTEQUALKEY, rather than the more commonly used status of *OKAY.

Example 2: Check a nominated tax code against a table of valid tax codes. The first key to the table indicates the tax classification which in this case is always income tax .

```
CHECK_FOR IN_FILE(TAXTAB) WITH_KEY('INCOME' #TAXCDE)

IF_STATUS IS_NOT(*EQUALKEY)
MESSAGE MSGTXT('Tax code specified not valid for income tax')
ENDIF
```

7.11 CLOSE

The CLOSE command is used to close the database file specified by the FILE parameter. Individual files can be closed, or all files in the function open at the time the command is issued can be closed.

Also See

[7.11.1 CLOSE Parameters](#)

[7.11.2 CLOSE Comments / Warnings](#)

[7.11.3 CLOSE Examples](#)

Optional

```
CLOSE ----- FILE----- *ALL   . *FIRST  -----  
>  
      file name . library name  
  
>-- IO_STATUS ---- *STATUS ----->  
      field name  
  
>-- IO_ERROR ----- *ABORT -----|  
      *NEXT  
      *RETURN  
      label
```

7.11.1 CLOSE Parameters

FILE

IO_ERROR

IO_STATUS

FILE

Specifies the file to be closed. Individual files can be selected or the default of *ALL can be nominated. For more information, refer to the section on [Specifying File Names in I/O Commands](#).

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

For values, refer to [I/O Return Codes](#).

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command. The purpose of *NEXT is to permit you to handle error messages in the RDML, and then ABORT, rather than use the default ABORT. (It is possible for processing to continue for LANSAs for i and Visual LANSAs, but this is NOT a recommended way to use LANSAs.)

ER returned from a database operation is a fatal error and LANSAs does not expect processing to continue. The IO Module is reset and further IO will be as

if no previous IO on that file had occurred. Thus you must not make any presumptions as to the state of the file. For example, the last record read will not be set. A special case of an IO_ERROR is when a trigger function is coded to return ER in TRIG_RETC. The above description applies to this case as well. Therefore, LANSAs recommends that you do NOT use a return code of ER from a trigger function to cause anything but an ABORT or EXIT to occur before any further IO is performed.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

7.11.2 CLOSE Comments / Warnings

- Refer to the OPEN command for more details of file OPEN and CLOSE command considerations.
- Normally there is no need to code specific OPEN or CLOSE commands into an RDML program. LANSAs will automatically open and close the file by default. The CLOSE command is only used when you wish to change the default method that LANSAs uses to open and close the file.

7.11.3 CLOSE Examples

Example 1: Close the customer master file CUSTMST.

```
CLOSE FILE(CUSTMST)
```

Example 2: Close all files which are active at this time.

```
CLOSE
```

Example 3: A temporary work file must be closed in order to clear the data.

```
CLOSE FILE(WORKFILE.QTEMP)  
EXEC_CPF COMMAND('CLRPFM FILE(QTEMP/WORKFILE)')
```

7.12 CLR_LIST

The CLR_LIST command is used to clear all entries from a list.

The list may be a **browse** list (used for displaying information at a workstation) or a **working** list (used to store information within a program).

Refer to the DEF_LIST command for more details of lists and list processing.

Also See

[7.12.1 CLR_LIST Parameters](#)

[7.12.2 CLR_LIST Example](#)

Optional

```
CLR_LIST ----- NAMED ----- *FIRST -----  
---|  
           name of list
```

7.12.1 CLR_LIST Parameters

NAMED

Specifies the name of the list which is to be cleared.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which may be a browse or a working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

7.12.2 CLR_LIST Example

The following example applies to the CLR_LIST command.

Clear all entries from a list named #ORDERLINE:

```
CLR_LIST NAMED(#ORDERLINE)
```

7.13 COMMIT

The COMMIT command is used to cause an operating system "commit" operation to be issued. A commit operation commits all uncommitted changes to the database.

Refer to the appropriate IBM supplied manual for details of IBM i commitment control processing before attempting to use this command.

It is also advisable to read [Commitment Control](#) in the *LANSA for i User Guide*.

Portability If using Visual LANSA, refer to [Commitment Control](#) in the
Considerations *LANSA Application Design Guide*.

Also See

[7.13.1 COMMIT Parameters](#)

[7.13.2 COMMIT Example](#)

COMMIT ----- no parameters -----

|

7.13.1 COMMIT Parameters

The COMMIT command has no parameters.

7.13.2 COMMIT Example

Request that the user input details of an order and then write the order header and all associated lines to the database before issuing a commit operation:

```
GROUP_BY  NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #  
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
```

```
SET_MODE  TO(*ADD)  
INZ_LIST  NAMED(#ORDERLINE) NUM_ENTRYYS(20)  
REQUEST   FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)
```

```
INSERT    FIELDS(#ORDERHEAD) TO_FILE(ORDHDR)  
SELECTLIST NAMED(#ORDERLINE) GET_ENTRYYS(*NOTNULL)  
INSERT    FIELDS(#ORDERLINE) TO_FILE(ORDLIN)  
ENDSELECT
```

```
COMMIT
```

If the function were to fail when writing the 4th order line (say), then an automatic rollback would be issued. This would cause the order header and any order lines already created to be rolled back from the file.

7.14 CONDCHECK

The CONDCHECK command is used to check a field by evaluating a condition or expression.

Also See

[7.14.1 CONDCHECK Parameters](#)

[7.14.2 CONDCHECK Comments / Warnings](#)

[7.14.3 CONDCHECK Examples](#)

[7.4 BEGINCHECK](#)

[7.7 CALLCHECK](#)

[7.16 DATECHECK](#)

[7.35 ENDCHECK](#)

[7.48 FILECHECK](#)

[7.78 RANGECHECK](#)

[7.99 VALUECHECK](#)

Required

```
CONDCHECK ---- FIELD ----- field name -----  
----->
```

```
>-- COND ----- condition ----->
```

Optional

```
>-- IF_TRUE ----- *NEXT ----->  
      *ERROR  
      *ACCEPT
```

```
>-- IF_FALSE ----- *ERROR ----->  
      *NEXT  
      *ACCEPT
```

```
>-- MSGTXT ----- *NONE ----->  
      message text
```

>-- *MSGID* ----- *DCU0004* ----->
 message identifier

>
>-- *MSGF* ----- *DC@M01 . *LIBL* -----
 message file . library name

>-- *MSGDTA* ----- *substitution variables* -----|
 | *expandable group expression* |
 ----- *20 max* -----

7.14.1 CONDCHECK Parameters

COND

FIELD

IF_FALSE

IF_TRUE

MSGID

MSGDTA

MSGF

MSGTXT

FIELD

Specifies the name of the field which is to be associated with the check.

COND

Specifies the condition or expression that is to be evaluated as either "true" or "false".

IF_TRUE

Specifies the action to be taken if the condition in the COND parameter is found to be true.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

IF_FALSE

Specifies the action to be taken if the condition in the COND parameter is found to be false.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match

in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAs aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA("BOLTS  " #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.14.2 CONDCHECK Comments / Warnings

- CONDCHECK commands must be within a BEGINCHECK / ENDCHECK validation block. Refer to these commands for further details.

7.14.3 CONDCHECK Examples

[Structuring Functions for Inline Validation](#)

[Structuring Functions to Use a Validation Subroutine](#)

[Using the CONDCHECK Command for Inline Validation](#)

[Using the CONDCHECK Command for Validation with a Subroutine](#)

Structuring Functions for Inline Validation

Typically functions using validation commands (eg: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for inline validation like this:

```
BEGIN_LOOP
REQUEST  << INPUT >>
BEGINCHECK
*        << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK
*        << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is passed back to the REQUEST command. This happens because of the default IF_ERROR(*LASTDIS) parameter on the ENDCHECK command.

Structuring Functions to Use a Validation Subroutine

Typically functions using validation commands (eg: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for subroutine validation like this:

```
DEFINE   FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')

BEGIN_LOOP
DUNTIL  COND(*NOERRORS)
REQUEST << INPUT >>
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
```

```
*      << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE  FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
*      << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is returned to the main function loop with #ERRORCNT > 0.

Using the CONDCHECK Command for Inline Validation

This example demonstrates how to use the CONDCHECK command within the main program block to check the value of a field against a set of conditions. Here the salary of the new employee is added to the current salary of the department and checked that it is still under the salary budget.

```
DEFINE  FIELD(#NEWSALARY) REFFLD(#SALARY) LABEL('New Sal
DEFINE  FIELD(#TOTSALARY) REFFLD(#SALARY) DEFAULT(0)
DEFINE  FIELD(#BUDGET) REFFLD(#SALARY) LABEL('Budget')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #NEWSALARY)

BEGIN_LOOP
REQUEST  FIELDS(#DEPARTMENT #BUDGET #EMPNO #NEWSALARY)
CHANGE  FIELD(#TOTSALARY) TO(*DEFAULT)

SELECT  FIELDS(#SALARY) FROM_FILE(PSLMST1) WITH_KEY(#DE
CHANGE  FIELD(#TOTSALARY) TO('#TOTSALARY + #SALARY')
ENDSELECT

BEGINCHECK
CONDCHECK FIELD(#NEWSALARY) COND('#NEWSALARY + #TOTS
ENDCHECK
```

```
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

If the salary for the new employee, when added to all existing salaries for that department, exceeds the budget for salaries the message defined with the CONDCHECK command is issued and program control returns to the last screen displayed. In this case the last screen displayed is the REQUEST screen.

Using the CONDCHECK Command for Validation with a Subroutine

This example demonstrates how to use the CONDCHECK command inside a subroutine to check the value of a field against a set of conditions.

After the user enters the requested details the VALIDATE subroutine is called. It checks that the salary of the new employee when added to all existing salaries for the department is still under the salary budget. If this condition is not true the message defined in the CONDCHECK command is given and the DOUNTIL loop executes again. When this condition is true the DOUNTIL loop ends and processing of the verified input is done.

```
DEFINE FIELD(#ERRORCNT) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
DEFINE FIELD(#NEWSALARY) REFFLD(#SALARY) LABEL('New Sal')
DEFINE FIELD(#TOTSALARY) REFFLD(#SALARY) DEFAULT(0)
DEFINE FIELD(#BUDGET) REFFLD(#SALARY) LABEL('Budget')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #NEWSALARY)
```

```
BEGIN_LOOP
DOUNTIL COND(*NOERRORS)
REQUEST FIELDS(#DEPARTMENT #BUDGET #EMPNO #NEWSALARY)
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
CHANGE FIELD(#TOTSALARY) TO(*DEFAULT)
SELECT FIELDS(#SALARY) FROM_FILE(PSLMST1) WITH_KEY(#DEPARTMENT)
CHANGE FIELD(#TOTSALARY) TO('#TOTSALARY + #SALARY')
ENDSELECT
```

```
BEGINCHECK KEEP_COUNT(#ERRORCNT)
CONDCHECK FIELD(#NEWSALARY) COND('(#NEWSALARY + #TOTS
ENDCHECK IF_ERROR(*NEXT)
```

```
ENDROUTINE
```

7.15 CONTINUE

The CONTINUE command is a loop modifying command. It causes the next iteration of the loop structure it is within to be processed immediately.

CONTINUE/LEAVE commands work inside all loop commands.

Also See

[7.15.1 CONTINUE Parameters](#)

[7.15.2 CONTINUE Comments / Warnings](#)

[7.15.3 CONTINUE Examples](#)

[7.3 BEGIN_LOOP](#)

[7.30 DUNTIL](#)

[7.31 DOWHILE](#)

[7.67 LEAVE](#)

[7.83 SELECT](#)

Required

CONTINUE ----->

Optional

>-- *IF* ----- '*condition*' -----|

7.15.1 CONTINUE Parameters

IF

Optionally specifies the condition that is to be evaluated to determine if the CONTINUE should be executed. If not specified the CONTINUE is executed immediately. For more details, refer to [Specifying Conditions and Expressions](#).

7.15.2 CONTINUE Comments / Warnings

- The CONTINUE loop modifying command only applies to the following RDML loop structures:- SELECT/ENDSELECT, SELECTLIST/ENDSELECT, SELECT_SQL/ENDSELECT, DOWHILE/ENDWHILE, DOUNTIL/ENDUNTIL and BEGIN_LOOP/END_LOOP.

The CONTINUE command operates as follows:-

-----> Begin loop command

|

----- CONTINUE

End loop command

7.15.3 CONTINUE Examples

Using CONTINUE within a BEGIN_LOOP loop

Using CONTINUE within a SELECT

Using CONTINUE within a BEGIN_LOOP loop

This example demonstrates how to use the CONTINUE command in a BEGIN_LOOP loop.

```
DEF_LIST  NAME(#EMPBROSWE) FIELDS(#EMPNO #SURNAME #GIV  
  
BEGIN_LOOP  
REQUEST  FIELDS(#EMPNO) BROWSELIST(#EMPBROSWE)  
FETCH    FIELDS(#EMPNO #SURNAME #GIVENAME #DEPARTMENT) FI  
  
IF_STATUS IS_NOT(*OKAY)  
MESSAGE  MSGTXT('That employee could not be found!')  
CONTINUE  
ENDIF  
  
ADD_ENTRY TO_LIST(#EMPBROSWE)  
END_LOOP
```

If the requested employee number is not found the message is issued and the CONTINUE command causes program control to skip the ADD_ENTRY command and return to the top of the loop at the REQUEST command.

Using CONTINUE within a SELECT

This example demonstrates how to use the CONTINUE command within a SELECT loop. Here, with the use of an additional user function key, selected records can be viewed and dropped if not required.

```
DEF_COND  NAME(*DROPPED) COND('#IO$KEY = "09"')  
DEF_LIST  NAME(#EMPBROWSE) FIELDS(#SECTION #EMPNO #SURNAME #GIVENAME)  
DEF_LIST  NAME(#EMPSELECT) FIELDS(#SECTION #EMPNO #SURNAME #GIVENAME)  
  
SELECT    FIELDS(#EMPBROWSE) FROM_FILE(PSLMST)  
DISPLAY   FIELDS(#SECTION #EMPNO #SURNAME #GIVENAME) BR  
CONTINUE  IF(*DROPPED)  
ADD_ENTRY TO_LIST(#EMPBROWSE)
```

```
ADD_ENTRY TO_LIST(#EMPSELECT)
ENDSELECT
DISPLAY BROWSELIST(#EMPSELECT)
```

7.16 DATECHECK

The DATECHECK command is used to check if a date field is valid against one of five possible formats and optionally that the date is within a certain number of days before and/or after the current date.

Also See

[7.16.1 DATECHECK Parameters](#)

[7.16.2 DATECHECK Comments / Warnings](#)

[7.16.3 DATECHECK Examples](#)

[7.4 BEGINCHECK](#)

[7.7 CALLCHECK](#)

[7.14 CONDCHECK](#)

[7.35 ENDCHECK](#)

[7.48 FILECHECK](#)

[7.78 RANGECHECK](#)

[7.99 VALUECHECK](#)

Required

DATECHECK ---- *FIELD* ----- *field name* -----
---->

Optional

>-- *IN_FORMAT* ---- **SYSFMT* -----
>

**DDMMYY*
**MMDDYY*
**YYMMDD*
**DDMMYYYY*
**YYYYMMDD*
**YYYYDDMM*
**YYMM*
**MMYY*
**MMDDYYYY*

**YYYYMM*
**MMYYYY*
**SYSFMT8*

>-- *BEFORE* ----- 9999999 ----->
numeric value

>-- *AFTER* ----- 9999999 ----->
numeric value

>-- *IF_VALID* ----- **NEXT* ----->
**ERROR*
**ACCEPT*

>-- *IF_INVALID* --- **ERROR* -----
>
**NEXT*
**ACCEPT*

>-- *MSGTXT* ----- **NONE* -----
>
message text

>-- *MSGID* ----- *DCU0006* -----
>
message identifier

>-- *MSGF* ----- *DC@M01 . *LIBL* -----
>
message file . library name

>-- *MSGDTA* ----- *substitution variables* -----|
| *expandable group expression* |
----- *20 max* -----

7.16.1 DATECHECK Parameters

AFTER

BEFORE

FIELD

IF_INVALID

IF_VALID

IN_FORMAT

MSGDTA

MSGF

MSGID

MSGTXT

FIELD

Specifies the name of the field which is to be checked. Only fields of type Alpha, Packed, Signed, Date, or Datetime may be specified. Fields of any other type cannot be specified.

IN_FORMAT

Specifies the format the date is expected in. The only permissible values are *SYSFMT, *DDMMYY, *MMDDYY, *YYMMDD, and *DDMMYYYY, *YYYYMMDD, *YYYYDDMM, *YYMM, *MMYY, *MMDDYYYY, *YYYYMM, *MMYYYY, *SYSFMT8.

*SYSFMT, which is the default value, indicates the date format nominated in the operating system "system value" QDATFMT should be used as the date format required. Refer to the appropriate IBM supplied manual for more details of system value QDATFMT.

If the specified field is of type Date or Datetime, this parameter is irrelevant as fields of type Date or Datetime are always in ISO format. So, if this parameter is specified as other than *SYSFMT, a FFC Warning is displayed stating that the format is ignored.

BEFORE

Specifies the number of days prior to the current date for which this date will still be valid. If no days are specified, the value 9999999 is assumed.

AFTER

Specifies the number of days after the current date for which this date will still be valid. If no days are specified, the value 9999999 is assumed.

IF_VALID

Specifies the action to be taken if the field is a valid date in the format nominated and also passes any associated range checking.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

IF_INVALID

Specifies the action to be taken if the date is not a valid date in the format nominated or fails any associated range checking.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as .

"&1 are out of stock ... reorder &2"

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this :

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA("BOLTS  " #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.16.2 DATECHECK Comments / Warnings

- All dates must have a four character year so that accurate comparisons and calculations can be performed. Where a two character year (e.g. DDMMYY, YYMMDD, MMY) is supplied the century value is retrieved from the system definition data area. The year supplied is compared to a year in the data area, if the supplied year is less than or equal to the comparison year then the less than year is used. If the supplied year is greater than the comparison year then the greater than year is used.

7.16.3 DATECHECK Examples

[Structuring Functions for Inline Validation](#)

[Structuring Functions to Use a Validation Subroutine](#)

[Using the DATECHECK Command for Inline Validation](#)

[Using the DATECHECK Command for Validation with a Subroutine](#)

Structuring Functions for Inline Validation

Typically functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for inline validation like this:

```
BEGIN_LOOP
REQUEST  << INPUT >>
BEGINCHECK
    << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK
    << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is passed back to the REQUEST command. This happens because of the default IF_ERROR(*LASTDIS) parameter on the ENDCHECK command.

Structuring Functions to Use a Validation Subroutine

Typically functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for subroutine validation like this:

```
DEFINE  FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND  NAME(*NOERRORS) COND('#ERRORCNT = 0')

BEGIN_LOOP
DUNTIL  COND(*NOERRORS)
REQUEST  << INPUT >>
EXECUTE  SUBROUTINE(VALIDATE)
ENDUNTIL
```

```
*      << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE   FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
*      << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is returned to the main function loop with #ERRORCNT > 0.

Using the DATECHECK Command for Inline Validation

This example demonstrates how to use the DATECHECK command within the main program block to check the validity of date fields.

```
DEF_LIST  NAME(#EMPBROWSE) FIELDS(#EMPNO #GIVENAME #SU

BEGIN_LOOP
REQUEST  FIELDS(#EMPNO #STARTDTE) BROWSELIST(#EMPBROW

BEGINCHECK
DATECHECK FIELD(#STARTDTE) IN_FORMAT(*DDMMYY) BEFORE(
ENDCHECK

ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

If the value of #STARTDTE is not in format of DDMMYY, is more than 30 days in the past or is in the future the message defined with the DATECHECK command is issued and program control returns to the last screen displayed. In this case the last screen displayed is the REQUEST screen.

Using the DATECHECK Command for Validation with a Subroutine

This example demonstrates how to use the DATECHECK command inside a subroutine to check the validity of date fields.

After the user enters the requested details the VALIDATE subroutine is called. It

checks that the value of #STARTDTE is in the DD/MM/YY format, is 0 days in the future and is not more than 30 days in the past. If this is not true the message defined in the DATECHECK command is given and the DOUNTIL loop executes again. When #STARTDTE is the correct format and value the DOUNTIL loop ends and processing of the verified input is done.

```
DEFINE FIELD(#ERRORCNT) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #STARTDTE)
```

```
BEGIN_LOOP
DUNTIL COND(*NOERRORS)
REQUEST FIELDS(#EMPNO #STARTDTE) BROWSELIST(#EMPBROWSE)
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
```

```
BEGINCHECK KEEP_COUNT(#ERRORCNT)
DATECHECK FIELD(#STARTDTE) IN_FORMAT(*DDMMYY) BEFORE(
ENDCHECK IF_ERROR(*NEXT)
```

```
ENDROUTINE
```

7.17 DEF_ARRAY

The DEF_ARRAY command is used to define an array structure within an RDML function.

The array created by this command can be used for:

1. Grouping up to 1000 individual identical fields into contiguous storage, thus allowing "indexed" references to be made to any one of the fields. To do this, use the OF_FIELDS parameter. This is the only option supported for RDMLX fields. Note that if an object is not enabled for RDMLX it is limited to 100 fields.
2. Overlaying a large field with an array structure, thus allowing "indexed" references to individual areas within the larger field. To do this, use the OVERLAYING parameter. This option is not supported when using an RDMLX field.

This command provides full array support to LANSAs applications.

However, it must be emphasized that the use of array constructs in database files is **NOT** considered to be the best long term strategy in terms of relational database design.

This facility is provided as a concession to reality. Array structures within files provide better system performance than fully normalized relational database designs.

However, the presence of this facility should not be construed in any way as encouraging the use of array structures in database designs.

Whenever hardware resources permit, a fully relational design will **always** yield a simpler, easier to use, easier to maintain and longer lasting solution.

Also See

[7.17.1 DEF_ARRAY Parameters](#)

[7.17.2 DEF_ARRAY Comments / Warnings](#)

[7.17.3 DEF_ARRAY Examples](#)

Required

```
DEF_ARRAY ---- NAME ----- #name of array -----  
----->
```

```
>-- INDEXES ----- #index field name ----->
```

| |
----- 50 max -----

>-- *OF_FIELDS* ---- #field name ----->
| expandable group expression |
----- 100 max -----

-- OR --

>-- *OVERLAYING* --- #field name -- start position --

>

Optional

>-- *TYPE* -----

- **CHAR* **DEC* **PACKED* **SIGNED* ----->

>-- *TOT_ENTRY* ---- total entries (1-9999) -----

>

>-- *ENTRY_LEN* ---- entry length (1-256) -----

>

>-- *ENTRY_DEC* ---- number of decimals (0-9) -----

--|

7.17.1 DEF_ARRAY Parameters

ENTRY_DEC

ENTRY_LEN

INDEXES

NAME

OF_FIELDS

OVERLAYING

TOT_ENTRY

TYPE

NAME

Specifies the name that is to be assigned to the array. The following warnings and comments apply to the assignment of array names:

- The name must begin with the standard "#" symbol.
- The name can be from 2 to 4 characters long (including the "#").
- The second character must be a letter from the English alphabet.
- Subsequent characters must be a letter from the English alphabet, or, one of the values 1,2,3,4,5,6,7,8,9,0 or \$.
- It is strongly recommended that array naming standards are devised for application systems to facilitate the exchange of array data between functions and all improved impact analysis abilities.
- The following names are reserved and must **not** be used
- any name containing an "@" symbol.
- any name containing an imbedded "#" symbol.
- names containing "MVR", "CMK", "RIM" or "GEN".
- 'TAB' can not be used as an array name.

INDEXES

Specifies the names of the numeric fields that will be used as indexes to reference individual elements in this array.

At least one index name must be specified. Up to 50 may be specified.

The following warnings and comments apply to the assignment of array index field names:

- The name must begin with the standard "#" symbol.
- The name must be 3 characters long (including the "#").
- The second character must be a letter from the English alphabet.
- The third character must be a letter from the English alphabet, or one of the values 1,2,3,4,5,6,7,8,9,0 or \$.
- The array index field must be a numeric field (packed or signed) and must be either defined in this function or in the LANSAs data dictionary.
- The array index field cannot be an RDMLX field.
- The following names are reserved and must **not** be used
- any name containing an "@" symbol.
- any name containing an imbedded "#" symbol.

It is strongly recommended that several standard array index names are defined in the data dictionary as used by all programs. A minimum suggested set of index fields would be:

Name	Type	Length	Decimals
II	P	7	0
JJ	P	7	0
KK	P	7	0
LL	P	7	0
MM	P	7	0
NN	P	7	0

Do not skimp on the number of digits to values of 2 or 3, this will only cause later problems.

Array index fields must not be overlaid on or by other fields (in any context).

OF_FIELDS

Specifies from 1 to 100 fields that are to be grouped into a contiguous area so that they can be referenced by index via this array.

All fields must have the same type, length and number of decimal positions. An

expandable group expression is allowed in this parameter. **Either none of them have the ASQN attribute, or ALL of them have the ASQN attribute.**

Any field specified in this parameter must **NOT** itself be overlaid onto another field by using the TO_OVERLAY parameter of DEFINE or OVERRIDE commands.

Once a field has been specified in the OF_FIELDS parameter of one DEF_ARRAY command it cannot be specified in the OF_FIELDS parameter of any other DEF_ARRAY command (ie: a field can only be defined in one array).

An RDMLX field does not have a #xxx#ARRAY field automatically defined, whether or not the length is less than or equal to 256.

Failure to observe these rules may cause unpredictable results.

The OF_FIELDS parameter and the OVERLAYING parameter are mutually exclusive. You must specify one, but cannot specify both.

OVERLAYING

Specifies the name of a field that is to be fully or partially overlaid by this array. RDMLX fields may not be overlaid, nor overlay another field.

When a full overlay is required, the total length (in bytes) of the array should match the total length (in bytes) of the field being overlaid.

When a partial overlay is required, the total length (in bytes) of the array may be less than the total length (in bytes) of the field being overlaid.

The optional start position component of this parameter allows the array to partially overlay only a component of the field being overlaid.

Whether a full or partial overlay is required, you must **NOT** define the array so that it goes beyond the last position of the field being overlaid. Failure to observe this rule will cause unpredictable results.

The field that is referenced by this parameter must **NOT** itself be overlaid onto another field by using the TO_OVERLAY parameter of the DEFINE or OVERRIDE commands.

The OVERLAYING parameter and the OF_FIELDS parameter are mutually exclusive. You must specify one, but cannot specify both.

TYPE

Specifies the type of data storage to be used within the array. Valid types are:

*CHAR Character or alphanumeric data is to be stored in the array. Corresponds to the LANSAs dictionary type of "A".

***PACKED/*DEC** Packed decimal data is to be stored in the array. Corresponds to the LANSAs dictionary type of "P".

***SIGNED** Signed or zoned decimal data is to be stored in the array. Corresponds to the LANSAs dictionary type of "S".

Do **NOT** specify this parameter when you use the OF_FIELDS parameter. Its value is automatically deduced from the type of the field(s) specified in the OF_FIELDS parameter.

TOT_ENTRY

Specifies the total number of entries the array will contain. Any integer value in the range 1 to 9999 is valid.

Do **NOT** specify this parameter when you use the OF_FIELDS parameter. Its value is automatically deduced from the number of field(s) specified in the OF_FIELDS parameter.

ENTRY_LEN

Specifies the total length of each individual entry in the array.

For type *CHAR arrays, the value must be an integer in the range 1 to 256.

For type *DEC, *PACKED and *SIGNED arrays, the value must be an integer in the range 1 to 30. The value you specify here is the total number of digits the numeric field contains. For packed fields this does **not** represent the number of bytes of memory each array entry will occupy.

Do **NOT** specify this parameter when you use the OF_FIELDS parameter. Its value is automatically deduced from the length of the field(s) specified in the OF_FIELDS parameter.

ENTRY_DEC

Specifies the total number of decimal positions (ie: digits, not bytes) that each numeric array entry should contain.

For type *CHAR arrays this value is ignored.

For type *DEC, *PACKED and *SIGNED arrays the value must be an integer in the range 0 to 9.

Do **NOT** specify this parameter when you use the OF_FIELDS parameter. Its value is automatically deduced from the number of decimal positions of the field(s) specified in the OF_FIELDS parameter.

7.17.2 DEF_ARRAY Comments / Warnings

To use the array facility properly you must understand the IBM i data storage formats of character, packed decimal and signed / zoned decimal.

When you define an array various fields are automatically defined into the function, just as if you had defined them yourself using the DEFINE command. RDMLX fields may not be overlaid, nor overlay another field.

The following example assumes that #VAL01, #VAL02 and #VAL03 are all packed decimal fields of length 7, with 2 decimals:

```
DEF_ARRAY NAME(#VAL) INDEXES(#II #JJ) OF_FIELDS(#VAL01 #VAI
```

will define the following "fields" into your function:

- #VAL#II as a packed 7,2 field. This field allows you to make indexed references to array #VAL using index #II.
- #VAL#JJ as a packed 7,2 field. This field allows you to make indexed references to array #VAL using index #JJ.

Additionally, references to #VAL#II or #VAL#JJ by data validation commands like RANGECHECK and SET_ERROR will cause an error to be set in the associated OF_FIELD field.

For example:

```
CHANGE FIELD(#II) TO(3)  
SET_ERROR FOR_FIELD(#VAL#II)
```

```
CHANGE FIELD(#JJ) TO(1)  
SET_ERROR FOR_FIELD(#VAL#JJ)
```

```
DISPLAY FIELDS(#VAL01 #VAL02 #VAL03)
```

will cause fields #VAL01 and #VAL03 to be displayed in reverse video because they have had their error flags turned on by the SET_ERROR commands.

- These element fields #VAL#II and #VAL#JJ can be referenced as individual fields in almost all commands. Specific places where they **cannot** be used include:

- On any screen panel. DISPLAY FIELDS(#VAL#II) in any form is invalid and will cause a compile failure. Likewise they cannot be placed in browse lists. However, they can be placed into working lists.
- In EXEC_OS400 or EXEC_CPF commands, use an intermediate work field instead. See the following examples for details.
- In debug mode ... #VAL#II cannot be shown directly by the debug facility.
- In database operations. The following code sections are **not** equivalent. The second operation will yield no result.

```
FETCH FIELDS(#VAL01) FROM_FILE(.....)
```

and CHANGE #II 1

```
FETCH FIELDS(#VAL#II) FROM_FILE(.....)
```

#VAL#ARRAY as a character field of length 12. This field is the full representation of array #VAL in character format. In this case 3 * P(7,2) uses 12 bytes of storage. This field is only automatically defined when the aggregate array length is less than or equal to 256 bytes.

This full array field #VAL#ARRAY is very useful because it can be used to:

- Pass an entire array to another function via the exchange list or to a 3GL program via a parameter. Of course the array in the other function will have to have the same name and be identical in all other respects. Also remember that the complete exchange list area is only 2K bytes.
- Display the contents of the entire array while in debug mode.
- Allow alphanumeric arrays to be initialized in one command.
- Display on screen panels or reports. However, if the actual array content is packed decimal data it may cause a workstation device failure.
- Placed into a working list, thus facilitating 2 dimensional array processing. One "index" is the working list entry number, the other is the actual array entry index.
- Since fields like #VAL#II, #VAL#JJ and #VAL#ARRAY are real fields in the program, it is possible to override their attributes with an OVERRIDE command. Do **not** override their lengths or number of decimal positions.
- When an RDML function is translated into RPG code, up to 40 arrays may be defined by LANSA to facilitate program processing. The number and type vary with the complexity of, and facilities used by, the RDML function.

- Additionally, every group, list, database operation or screen panel interaction will cause additional arrays to be defined.
- An RPG program is limited to having 200 arrays.
- So while the actual number that any RDML program will consume automatically is impossible to predict until compile time, you should plan on not ever defining more than 50 to 100 arrays in any individual function. In a single RDML function, number of arrays multiplied by the number of indexes must be less or equal to 100.

7.17.3 DEF_ARRAY Examples

These examples apply to the DEF_ARRAY command.

Example 1: The data dictionary contains 3 packed decimal fields of length 7 with 2 decimals called VAL01, VAL02 and VAL03.

Group them all into an array called #VAL that will be indexed by field #II:

```
DEF_ARRAY NAME(#VAL) INDEXES(#II) OF_FIELDS(#VAL01 #VAL02
```

Example 2: Using array #VAL, request that the user inputs all 3 values, then check that they are all in the range 7 to 42:

```
DEF_ARRAY NAME(#VAL) INDEXES(#II) OF_FIELDS(#VAL01 #VAL02
```

```
REQUEST FIELDS(#VAL01 #VAL02 #VAL03)
```

```
BEGINCHECK
```

```
  BEGIN_LOOP FROM(1) TO(3) USING(#II)
```

```
  RANGECHECK FIELD(#VAL#II) RANGE((7 42))
```

```
  END_LOOP
```

```
ENDCHECK
```

The example demonstrates how the data validation commands VALUECHECK, RANGECHECK, FILECHECK, DATECHECK, CONDCHECK, CALLCHECK and SET_ERROR not only set an error for the array entry (#VAL#II) but also for the underlying field #VAL01, #VAL02 or #VAL03 (depending on the current value of #II).

Example 3: Using array #VAL, request the user inputs all 3 values, increment all values by 10 % and show the total to the user:

```
DEF_ARRAY NAME(#VAL) INDEXES(#II) OF_FIELDS(#VAL01 #VAL02
```

```
REQUEST FIELDS(#VAL01 #VAL02 #VAL03)
```

```
CHANGE FIELD(#RESULT) TO(0)
```

```
BEGIN_LOOP FROM(1) TO(3) USING(#II)
```

```
CHANGE FIELD(#VAL#II) TO('#VAL#II * 1.10')
```

```
CHANGE FIELD(#RESULT) TO('#RESULT + #VAL#II')
```

```
END_LOOP
```

```
DISPLAY FIELDS(#RESULT)
```

Example 4: The data dictionary contains an alphanumeric field of length 50 called #LIBLST. It is actually an "array" of up to 5 library names stored in one long field.

Define an array so that individual library names within it can be easily referenced:

```
DEF_ARRAY NAME(#LIB) TYPE(*CHAR) TOT_ENTRY(5) ENTRY_LEN
```

Example 5: Write a program so that field #LIBLST can be fetched from file USERDET, displayed on the screen as 5 separate fields, validated, and then updated back into the database:

```
DEF_ARRAY NAME(#LIB) TYPE(*CHAR) TOT_ENTRY(5) ENTRY_LEN  
INDEXES(#JJ) OVERLAYING(#LIBLST 1)
```

```
DEFINE FIELD(#LIB01) TYPE(*CHAR) LENGTH(10) TO_OVERLAY(#LI  
DEFINE FIELD(#LIB02) REFFLD(#LIB01) TO_OVERLAY(#LIBLST 11)  
DEFINE FIELD(#LIB03) REFFLD(#LIB01) TO_OVERLAY(#LIBLST 21)  
DEFINE FIELD(#LIB04) REFFLD(#LIB01) TO_OVERLAY(#LIBLST 31)  
DEFINE FIELD(#LIB05) REFFLD(#LIB01) TO_OVERLAY(#LIBLST 41)  
DEFINE FIELD(#LIBWRK) REFFLD(#LIB01)
```

```
FETCH FIELDS(#LIBLST) FROM_FILE(USERDET) WITH_KEY(*USER)  
REQUEST FIELDS(#LIB01 #LIB02 #LIB03 #LIB04 #LIB05)  
BEGINCHECK  
BEGIN_LOOP FROM(1) TO(5) USING(#JJ)  
  IF COND('#LIB#JJ *NE *BLANKS')  
    CHANGE FIELD(#LIBWRK) TO(#LIB#JJ)  
    EXEC_OS400 COMMAND('CHKOBJ QSYS/#LIBWRK *LIB') IF_ERROR  
    GOTO L20  
  L10: SET_ERROR FOR_FIELD(#LIB#JJ)  
  L20: ENDIF  
END_LOOP  
ENDCHECK  
UPDATE FIELDS(#LIBLST) IN_FILE(USERDET)
```

You can see from this example that there would have been less code in the program if field #LIBLST in file USERDET had actually been defined in the

file as five fields called #LIB01 -> #LIB05.

You will also note that EXEC_OS400 is one of the very few commands that will not accept indexed field references, which is why the work field #LIBWRK is required.

Example 6: Write a program so that field #LIBLST can be fetched from file USERDET and then printed on a report as one column:

```
DEF_ARRAY NAME(#LIB) TYPE(*CHAR) TOT_ENTRY(5) ENTRY_LEN

OVERRIDE FIELD(#LIB#JJ) COLHDG('Library' 'Names')

DEF_LINE NAME(#LINE01) FIELDS(#LIB#JJ)

FETCH FIELDS(#LIBLST) FROM_FILE(USERDET) WITH_KEY(*USER)

BEGIN_LOOP FROM(1) TO(5) USING(#JJ)
  IF COND('#LIB#JJ *NE *BLANKS')
    PRINT LINE(#LINE01)
  ENDIF
END_LOOP
ENDPRINT
```

Example 7: A SALES file contains one field called SALDATA that consists of MONTHS (as a 12 * character (2) array), and then EXPECTED SALES and ACTUAL SALES (as two separate 12 * packed decimal (7,2) decimal arrays). Define arrays that will allow indexed references to any component of the 3 arrays imbedded in this large field.

```
DEF_ARRAY NAME(#MTH) TYPE(*CHAR) TOT_ENTRY(12) ENTRY_LEN
DEF_ARRAY NAME(#EXP) TYPE(*DEC) TOT_ENTRY(12) ENTRY_LEN
DEF_ARRAY NAME(#ACT) TYPE(*DEC) TOT_ENTRY(12) ENTRY_LEN
```

Example 8: Using the file from example 7, print all records from the file in columns and produce grand totals:

```
DEF_ARRAY NAME(#MTH) TYPE(*CHAR) TOT_ENTRY(12) ENTRY_LEN
OVERRIDE FIELD(#MTH#II) COLHDG('Month')
DEF_ARRAY NAME(#EXP) TYPE(*DEC) TOT_ENTRY(12) ENTRY_LEN
OVERRIDE FIELD(#EXP#II) COLHDG('Expected' 'Sales') EDIT_CODE(3)
DEF_ARRAY NAME(#ACT) TYPE(*DEC) TOT_ENTRY(12) ENTRY_LEN
```

```

OVERRIDE FIELD(#ACT#II) COLHDG('Actual' 'Sales') EDIT_CODE(3)
DEF_LINE NAME(#SALES) FIELDS(#MTH#II #EXP#II #ACT#II)
DEFINE FIELD(#EXP_TOT) REFFLD(#EXP#II) LABEL('Total Expected')
DEFINE FIELD(#ACT_TOT) REFFLD(#ACT#II) LABEL('Total Actual')
DEF_BREAK NAME(#TOTAL) FIELDS(#EXP_TOT #ACT_TOT)
SELECT FIELDS(#SALDATA) FROM_FILE(SALES)
  BEGIN_LOOP FROM(1) TO(12) USING(#II)
  KEEP_TOTAL OF_FIELD(#EXP#II) IN_FIELD(#EXP_TOT)
  KEEP_TOTAL OF_FIELD(#ACT#II) IN_FIELD(#ACT_TOT)
  PRINT LINE(#SALES)
  END_LOOP
ENDSELECT
ENDPRINT
* automatically prints grand total lines

```

7.18 DEF_BREAK

The DEF_BREAK command is used to define one or more break lines for inclusion on a report.

Only fields of type Alpha, Packed and Signed may be specified. RDMLX field types cannot be specified.

Break lines are only printed when the condition specified in the TRIGGER_BY parameter is true. Break lines are primarily intended to produce level or control "break" lines in a report.

You should read [Producing Reports Using LANSAs](#) in the *Developers Guide* before attempting to use the DEF_BREAK command.

Also See

[7.18.1 DEF_BREAK Parameters](#)

[7.18.2 DEF_BREAK Comments / Warnings](#)

[7.18.3 DEF_BREAK Example](#)

Required

```
DEF_BREAK ---- NAME ----- name of break group ----  
----->
```

Optional

```
>-- FIELDS ----- field name field attributes --->  
      |           |           ||  
      |           --- 7 max ----- |  
      | expandable group expression |  
      ----- 100 max -----  
  
>-- TRIGGER_BY --- *DEFAULT -----  
-->  
      list of field names  
      | expandable group expression|  
      -----20 max -----  
  
>-- TYPE ----- *TRAILING ----->
```

**LEADING*

>-- *SPACE_BEF* ---- 2 ----->
decimal value

>-- *SPACE_AFT* ---- 1 ----->
decimal value

>-- *TEXT* ----- 'text' --- line/ --- position -->
 | row column |
 ----- 50 max -----
 **TMAPnnn 1 1 (special value)*

>-- *FOR_REPORT* --- 1 ----->
report number 1 -> 8

>-- *DESIGN* ----- **ACROSS* ----->
**DOWN*

>-- *IDENTIFY* ----- **LABEL* ----->
**COLHDG*
**NOID*

>-- *DOWN_SEP* ---- 1 ----->
decimal value

>-- *ACROSS_SEP* --- 1 ----->
decimal value

>-- *HEAD_COND* ---- **NONE* -----
->
name of condition

>-- *SUBROUTINE* --- **NONE* -----
-|
name of subroutine

7.18.1 DEF_BREAK Parameters

ACROSS_SEP

DESIGN

DOWN_SEP

FIELDS

FOR_REPORT

HEAD_COND

IDENTIFY

NAME

SPACE_AFT

SPACE_BEF

SUBROUTINE

TEXT

TRIGGER_BY

TYPE

NAME

Specifies the name that is to be assigned to the group of report print lines defined by this command. The name specified must be unique within the function.

FIELDS

Specifies the field(s) that is to be printed on the report. An expandable group expression is allowed in this parameter.

Only RDML fields are supported.

TRIGGER_BY

Specifies the condition that is to be used to "trigger" the printing of the break line(s) defined by this command. An expandable group expressions is allowed in this parameter.

*DEFAULT, which is the default value, indicates that the break line should only be triggered once, when the report is finished / closed by the ENDPRINT command or the termination of the function. This value is typically used to produce "grand" total break lines at the end of a report.

Otherwise, specify a list of field names that are to be used to trigger the printing

of the break line(s). Every time **any** report line is printed LANSAs compares the fields nominated in the list with their previous values. If **any field** nominated in the list has changed value the break line(s) will be produced.

TYPE

Specifies the type of break line required.

*TRAILING, which is the default value, is typically used to produce break lines that trail after their associated detail lines.

*LEADING is typically used to produce break lines that are printed before their associated detail lines.

The difference between leading and trailing break lines is best illustrated by example. Consider the following detail (see DEF_LINE command) and trailing break line definitions:

```
DEF_LINE NAME(#DETAIL) FIELDS(#REGION #PRODES #VALUE)
```

```
DEF_BREAK NAME(#REGTOT) FIELDS(#REGVAL) TRIGGER_BY(#RE
```

A report produced using these line definitions might look like this:

Region	Product description	Value
NSW	Tinned goods	400.00
NSW	Paper plates	700.00
NSW	Plastic spoons	300.00
	Region total	1400.00
VIC	Tinned goods	500.00
VIC	Paper plates	750.00
VIC	Plastic spoons	100.00
	Region total	1350.00

In this example the line Region total 1400.00 is the trailing break line.

The use of the break lines as typical "trailing" subtotals can be seen. However,

the appearance of the report can be improved by using a "leading" break line as well:

```
DEF_LINE NAME(#DETAIL) FIELDS(#PRODES #VALUE)
```

```
DEF_BREAK NAME(#REGTOT) FIELDS(#REGVAL) TRIGGER_BY(#RE
```

```
DEF_BREAK NAME(#REGNAM) FIELDS(#REGION) TRIGGER_BY(#RE
```

When the report is produced now it would probably look something like this:

Product Description	Value
Region NSW	
Tinned goods	400.00
Paper plates	700.00
Plastic spoons	300.00
Region total	1400.00
Region VIC	
Tinned goods	500.00
Paper plates	750.00
Plastic spoons	100.00
Region total	1350.00

In this example the line Region NSW is the leading break line and the line Region total 1400.00 is the trailing break line.

SPACE_BEF

Specifies the number of lines on the report that should be spaced before the break line(s) is printed. The default value is 2, but any value in the range 0 to 100 can be specified.

SPACE_AFT

Specifies the number of lines on the report that should be spaced after the break line(s) is printed. The default value is 1, but any value in the range 0 to 100 can be specified.

TEXT

Allows the specification of up to 50 "**text strings**" that are to appear on the screen panel or report. Each text string specified is restricted to a maximum length of 20 characters.

When a text string is specified it should be followed by a row/line number and a column/position number that indicates where it should appear on the screen panel or report.

For example:

```
TEXT(('ACME' 6 2)('ENGINEERING' 7 2))
```

specifies 2 text strings to appear at line 6, position 2 and line 7, position 2 respectively.

Portability In Visual LANSA this parameter should only be edited using
Considerations the screen or report painter which will replace any text with a text map. DO NOT enter text using the command prompt or free format editor as it will not pass the full function checker if checked in to LANSA for i.

All Platforms

The text map is used by the screen or report design facilities to store the details of all the text strings associated with the screen panel or report lines.

Once a screen or report layout has been "painted" and saved, all text details from the layout are stored in a "text map". The text map is then subsequently changed by using the "painter" again.

The presence of a text map is indicated by a TEXT parameter that looks like this example:

```
TEXT((*TMAPnnn 1 1))
```

where "nnn" is a unique number (within this function) that identifies the stored

text map.

Some **very important things** about "text maps" and *TMAPnnn identifiers that you **must** know are:

- Never specify *TMAPnnn identifiers of your own or change *TMAPnnn identifiers to other values. Leave the assignment and management of *TMAPnnn identifiers to the screen and report design facilities.
- When copying a command that has an *TMAPnnn identifier, remove the *TMAPnnn references (ie: the whole TEXT parameter) from the copied command. If you fail to do this, then the full function checker will detect the duplicated use of *TMAPnnn identifiers, and issue a fatal error message before any loss occurs.
- Never remove an *TMAPnnn identifier from a command. If this is done then the associated text map may be deleted, or reused in another command, during a full function check or compilation. Loss of text details is likely to result.
- Never "comment out" a command that contains a valid *TMAPnnn identifier. This is just another variation of the preceding warning and it runs the same risks of loss or reuse of text.
- Never specify *TMAPnnn values in an Application Template. In the template context *TMAPnnn values have no meaning. Use the "text string" format in commands used in, and initially generated by, Application Templates.

FOR_REPORT

Specifies the report with which this command should be associated. Up to 8 reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this report is report number 1.

DESIGN

Specifies the design/positioning method which should be used for fields that **do not have specific positioning attributes** associated with them.

*ACROSS, which is the default value for the DEF_BREAK command, indicates that fields should be designed "across" the report line (ie: one after another).

*DOWN indicates that the fields should be designed "down" the report page (ie: one under another).

IDENTIFY

Specifies the default identification method to be used for fields that **do not have**

specific identification attributes associated with them.

*LABEL, which is the default value for the DEF_BREAK command, indicates that fields should be identified by their associated labels.

*COLHDG indicates that fields should be identified by their associated column headings.

*NOID indicates that no identification of the field is required. Only the field itself should be included into the report line(s).

DOWN_SEP

Specifies the spacing between lines on the report that should be used when automatically designing a report. The value specified must be a number in the range 1 to 10. The default value for the DEF_BREAK command is 1.

ACROSS_SEP

Specifies the spacing between columns on the report that should be used when automatically designing a report. The value specified must be a number in the range 0 to 10. The default value for the DEF_BREAK command is 1.

HEAD_COND

Optionally specifies the name of a condition that indicates whether any column heading line(s) associated with fields in **this** break print line are to be printed in the header area of the report.

*NONE, which is the default value, indicates that no controlling condition applies, and any column headings associated with this break line should **always** be printed in the report header area, regardless of which line is actually being printed.

If a controlling condition is specified, it must be defined elsewhere in the RDML function by a DEF_COND (define condition) command. At the time that **any** print line is to be printed the status of the condition will be checked. Only when it is found to be true will the column headings associated with this break print line be included in the header area of the report.

SUBROUTINE

Optionally specifies the name of a subroutine that is to be executed just prior to printing the break line.

*NONE, which is the default value, indicates that no subroutine should be executed before printing the break line.

If a subroutine name is specified, it **must**:

- Be defined within this function as a valid subroutine by using the SUBROUTINE command.
- Not have any parameters. Subroutines used this way cannot have parameters.

Any subroutine used with a break line should:

- Restrict itself to simple manipulations of fields that are to be printed on the break line. Fields other than those that appear on the print line may be changed but not in ways that are expected to communicate information to other parts of the RDML function at some later time.
- Avoid executing other subroutines.
- Avoid any screen panel interactions.
- Avoid printing any type of information at all.

These guidelines are not checked, but failure to observe them may lead to unpredictable results.

The logic used for invoking a **TYPE(*LEADING)** break line subroutine works like this:

```
<<if first usage or trigger values have changed>>
    <<execute leading break subroutine>>
    <<print the leading break line>>
<<endif>>

<<store current trigger values for next comparison>>
```

An example of using a subroutine with a **TYPE(*LEADING)** break line is as follows:

```
DEF_BREAK NAME(#REGION) FIELDS(#REG_CODE #REG_NAME) TF
SUBROUTINE NAME(GET_REGION)
FETCH FIELDS(#REG_NAME) FROM_FILE(REGIONS) WITH_KEY(#RE
ENDROUTINE
```

The leading break line is printed whenever a new region code is encountered.

The break subroutine uses the current region code value to extract the associated region name from the regions table. This is an efficient approach because the subroutine is only executed when a change of region code occurs.

The logic used for invoking a **TYPE(*TRAILING)** break line subroutine is slightly more complex because the values printed on a trailing break line actually "trail behind" the apparent (and visible) values in the RDML function.

For instance, a trailing break line that is triggered by a change of region code, and also prints the region name, is set up so that the printed region codes and region names "trail behind" the current values of the region code and region name fields visible in the RDML function.

The reason for this is simple.

When the region code changes from "001" to "002", say, the trailing break line is triggered. If it printed the current value of the region code then it would show as "002" on the report.

So a special internal "trailing" field containing the "previous" value of region code is actually printed, which still contains the value "001".

This "trailing" logic is applied to all fields that are printed on TYPE(*TRAILING) break lines.

While this may appear complex, it actually makes the RDML level logic for the developer much simpler, because the complexity of "trailing" logic is catered for internally and need not concern the developer.

The trailing break print logic goes like this:

```
<<if this is not first usage and trigger values have changed>>  
  
    <<save current values of all fields used in function>>  
  
    <<restore all printed fields from their "trailing" values>>  
  
    <<execute trailing break subroutine>>  
  
    <<move all printed fields back into their "trailing" values>>  
  
    <<restore current values of all fields used in function>>  
  
    <<print the trailing break line (ie: the trailing values)>>  
  
<<endif>>
```

<<store current trigger values for next comparison>>

At the completion of this logic all fields are restored back to what they were at the time that the logic was invoked.

This makes it effectively impossible for a trailing break line subroutine to "communicate" with other parts of the RDML function by changing field values.

An example of using a subroutine with a **TYPE(*TRAILING)** break line is as follows:

```
DEF_BREAK NAME(#REG_TOTAL) FIELDS(#REG_CODE #REG_NAME)

SELECT  FIELDS(#REG_CODE ... etc ...) FROM_FILE(SALES)
KEEP_TOTAL OF_FIELD(...) IN_FIELD(#REG_TOT1) BY_FIELD(#REG_
KEEP_TOTAL OF_FIELD(...) IN_FIELD(#REG_TOT2) BY_FIELD(#REG_
PRINT  LINE(*BREAKS) <-
ENDSELECT <-

SUBROUTINE NAME(REG_TOTAL)
FETCH FIELDS(#REG_NAME) FROM_FILE(REGIONS) WITH_KEY(#REG_
CHANGE FIELD(#REG_PCT) TO('(#REG_TOT1 / #REG_TOT2) * 100')
ENDROUTINE <-
```

Like the preceding example the subroutine is only invoked when a region code changes. It extracts the region name from the region table and also sets #REG_PCT to be the percentage of totalled fields #REG_TOT1 and #REG_TOT2.

This routine will work well because it only uses and changes values that are actually printed on the break line.

This approach would be apparent if you ran this application in debug mode.

If you set up this application to stop in debug mode at:

- The PRINT LINE(*BREAKS) command
- The ENDSELECT command
- The ENDROUTINE command (of the REG_TOTAL subroutine)

Then assuming that the SALES file contained information for region codes

"001" and "002", and that the SELECT loop had already processed all the region "001" information, and just read in the **first region "002" record** you would find:

At the PRINT command:

- Field #REG_CODE would contain "002".
- Field #REG_NAME would be blank.
- Field #REG_TOT1 would be the current total for region "002"
- Field #REG_TOT2 would be the current total for region "002"
- Field #REG_PCT would be zero.

At the ENDROUTINE command:

- Field #REG_CODE would contain "001".
- Field #REG_NAME would contain the name of region "001".
- Field #REG_TOT1 would be the final total for region "001"
- Field #REG_TOT2 would be the final total for region "001"
- Field #REG_PCT would be the region "001" percentage total.

At the ENDSELECT command:

- Field #REG_CODE would contain "002".
- Field #REG_NAME would be blank.
- Field #REG_TOT1 would be the current total for region "002"
- Field #REG_TOT2 would be the current total for region "002"
- Field #REG_PCT would be zero.

7.18.2 DEF_BREAK Comments / Warnings

- When assigning specific line attributes to fields or text in a DEF_BREAK command note that the line numbers used are "offsets" from the start of the print line. Thus specifying *L001 against a field does not mean the field will actually print on line 1 of the report. The field will print on line 1 of the "group" of fields that make up the DEF_BREAK command.
- If you use an expandable group expression in a DEF_BREAK command FIELDS and/or TRIGGER_BY parameter(s) and you change the layout using the report design facility, LANSAs will substitute the expression with the actual fields. This is the only way LANSAs can assign attributes to the individual fields regardless of the group they initially came from.
- Note that trailing breaks really do "**trail behind**" the data being processed by the function. For example, when a trailing break definition like this is "triggered" for printing:

```
DEF_BREAK NAME(#REGTOT) FIELDS(#REGION #REGVAL) TRIGGER
```

it does not really print the current content of fields #REGION and #REGVAL. It actually prints the contents of 2 "internal" fields that contain the "previous" #REGION value and the "previous" #REGVAL value. This feature is what makes the processing of "break" totalling so simple and quick within LANSAs.

7.18.3 DEF_BREAK Example

This example applies to the DEF_BREAK command.

Write an RDML program to read a regional sales file, print details of each record read and produce regional subtotals.

```
DEF_LINE  NAME(#DETAIL) FIELDS(#REGION #PRODES #VALUE)
DEF_BREAK NAME(#REGTOT) FIELDS(#REGVAL) TRIGGER_BY(#RE
```

```
SELECT  FIELDS(#DETAIL) FROM_FILE(SALEHIST)
KEEP_TOTAL OF_FIELD(#VALUE) IN_FIELD(#REGVAL) BY_FIELD(#R
PRINT  LINE(#DETAIL)
ENDSELECT
```

```
ENDPRINT
```

Refer also to [Producing Reports Using LANSAs](#).

7.19 DEF_COND

The DEF_COND command is used to define a condition that may be used for **one or more** of the following tasks:

- To "pre-define" an expression that is used repeatedly in subsequent IF, DOWHILE, DOUNTIL or CONDCHECK commands.
- To define a condition to be used to control the enabling (or disabling) of function keys on subsequent DISPLAY, REQUEST or POP_UP commands.
- To define a condition to be used to control the appearance (or non-appearance) of fields on a screen panel or a report line.

Also See

[7.19.1 DEF_COND Parameters](#)

[7.19.2 DEF_COND Examples](#)

Mandatory

```
DEF_COND ----- NAME ----- name of condition -----  
----->
```

```
>-- COND - 'condition to evaluate' ----->
```

Optional

```
>-- COLHDG ----- *YES -----|  
                  *NO
```

7.19.1 DEF_COND Parameters

NAME

COND

COLHDG

NAME

Specifies the name of the condition.

Every condition must have a name and it **must** start with an "*" (asterisk). The second character **must** be a letter of the alphabet. Any subsequent characters **must** be in the range A -> Z or 0 -> 9. This rule therefore prohibits the use of condition names that contain imbedded blanks.

Condition names starting with *R, *P, *L or *C followed by 1, 2 or 3 digits must not be used on fields to be displayed on screen panels or used in reports. This will cause a conflict with the field positioning attributes, for example, *R12, *L123, *C01 etc.

These condition names should also not be used: *COLUMN, *COL, *COLHEAD, *COLHDG, *LAB, *LABEL, *NOID, *NOIDENT, *NEWPAGE, *NEWFORMAT, *DESC, *DES.

Also these names are reserved and should not be used: *EQ, *NE, *LT, *LE, *GT, *GE, *AND, *OR, *LIKE, *EQU, *NEU, *LTU, *LEU, *GTU, *GEU and *LIKEU.

Names that conflict with system or multilingual variable names should also be avoided.

The maximum length allowed for a condition name (including the "*") is 10 characters.

Up to 99 conditions can be defined within a program. Every one must have a **unique** name.

Conditions should be defined before any of the other commands that reference them.

Since the DEF_COND command "defines" something, and is thus strictly non-executable, it is customary for it to be placed at the beginning of the program along with all other definition style commands (eg: DEFINE, DEF_LIST, GROUP_BY, etc).

COND

Specifies the condition that is to be evaluated to test the "truth" of the condition. For more details, refer to [Specifying Conditions and Expressions](#).

COLHDG

Specifies whether or not field column headings associated with a field controlled by this condition are to also be conditioned.

When a field in a print line is printed or a field in a browse list is displayed, you can control whether or not its associated column headings appear by using this parameter.

*YES, which is the default value, indicates that any associated column headings are subject to the condition just like the associated field.

*NO, if used, specifies that no condition controls any associated column headings. The column headings will be printed or displayed according to the normal rules.

For example, making a whole column disappear:

```
DEF_COND NAME(*AUTSAL) COND('#GROUP = HO') COLHDG(*YES)

DEF_LINE NAME(#DETAIL) FIELDS(#A #B #C (#SALARY *AUTSAL))
```

specifies that field #SALARY and its column headings are only to appear on the report if the #GROUP = HO. This means that if the group is not HO, the salary column effectively disappears from the report.

However, making a field selectively disappear:

```
DEF_COND NAME(*PRTAMT) COND('(#CREDLMT *EQ 0) *OR (#OVEF

DEF_LINE NAME(#DETAIL) FIELDS(#A #B #C (#AMTDUE *PRTAMT) #
```

specifies that field #AMTDUE is only to be printed if the credit limit is zero or the amount has been outstanding more than 30 days. However, the associated column headings **always appear** on the report.

Please note: A condition controlling a column heading is tested at the time the column heading is printed or displayed. In the case of a report this is at the time the first line on the report page is printed. In the case of a display, this is the time at which the DISPLAY, REQUEST or POP_UP command is executed.

Also: A condition controlling a field in a browse list is only tested at the time the browse list entry is added or updated.

Additionally: Attempting to condition field(s) in a DEF_BREAK (define break) line of type *TRAILING may produce confusing or misleading results because the line is implicitly "trailing" behind the program in terms of the data it is processing.

Finally: Even if all fields on a print line or in a browse list entry have been conditioned not to appear, they will still cause a "blank" line to appear on the report or display panel.

7.19.2 DEF_COND Examples

Example 1: Define the condition "A is less than B multiplied by C" so that it can be used repeatedly in other commands:

```
DEF_COND NAME(*ALTBC) COND('#A *LT (#B * #C)')
```

```
IF    COND(*ALTBC)  
ENDIF
```

```
DOWHILE COND(*ALTBC)  
ENDWHILE
```

```
DOUNTIL COND(*ALTBC)  
ENDUNTIL
```

```
BEGINCHECK  
CONDCHECK FIELD(#A) COND(*ALTBC) MSGTXT('A must be not be les  
ENDCHECK
```

Example 2: Use the DEF_COND command so that the CHANGE and DELETE function keys in a DISPLAY command are only enabled when the user's name is FRED or MARY or BILL:

```
DEF_COND NAME(*AUTHORISE)  
COND('#USER = FRED) *OR (#USER = MARY) *OR (#USER = BILL)')
```

```
DISPLAY FIELDS(.....) CHANGE_KEY(*YES *NEXT *AUTHORISE)  
DELETE_KEY(*YES *NEXT *AUTHORISE)
```

Example 3: Use the CHECK_AUTHORITY Built-In Function to generalise the previous example into an enquire and maintain program using a product master file called PRODMST:

```
DEF_COND NAME(*ALLOWCHG) COND('#CHANGE = Y')  
DEF_COND NAME(*ALLOWDLT) COND('#DELETE = Y')
```

```
USE    BUILTIN(CHECK_AUTHORITY) WITH_ARGS(PRODMST ""*LII
```

```
USE    BUILTIN(CHECK_AUTHORITY) WITH_ARGS(PRODMST ""*LII
```

```
REQUEST FIELDS(.....)
```

```
FETCH FIELDS(.....) FROM_FILE(PRODMST) WITH_KEY(.....)
```

```
DISPLAY FIELDS(.....) CHANGE_KEY(*YES *NEXT *ALLOWCHG) I
```

```
IF_MODE IS(*CHANGE)
```

```
UPDATE FIELDS(.....) IN_FILE(PRODMST)
```

```
ENDIF
```

```
IF_MODE IS(*DELETE)
```

```
DELETE FROM_FILE(PRODMST)
```

```
ENDIF
```

This is an effective way to safely and simply combine an enquiry program and maintenance program into one program. The CHANGE and DELETE function keys will only ever be enabled when the user is authorised to change or delete information, so no further program checking is required.

Example 4: Prevent field #SALARY from appearing on a screen panel unless the department number is 464:

```
DEF_COND NAME(*HEADOFF) COND('#DEPARTMENT = "464")
```

```
DISPLAY FIELDS(#A #B #C (#SALARY *HEADOFF) #E #F #G)
```

Example 5: Simplify complex conditions by breaking into smaller parts:

```
DEF_COND NAME(*SELCUST) COND('((#CUSTTYPE = "A") *OR (#CU
```

The previous condition can be made more maintainable and understandable by:

```
DEF_COND NAME(*SELTYPE) COND('(#CUSTTYPE = "A") *OR (#CUS
```

```
DEF_COND NAME(*SELBAL) COND('(#BALANCE *GT 10000) *AND (
```

```
DEF_COND NAME(*SELSALE) COND('#LASTSALE > #MONTHSTRT')
```

```
DEF_COND NAME(*SELCUST) COND('*SELTYPE *AND *SELBAL *A
```

Conditions can be used individually, or as part of a larger expression:

```
IF COND(*SELCUST)
DOWHILE COND('*SELCUST *OR (#ACTIVE *EQ "Y")')
```

7.20 DEF_FOOT

The DEF_FOOT command is used to define one or more foot lines for inclusion on a report.

Only fields of type Alpha, Packed and Signed may be specified. RDMLX field types cannot be specified.

Foot lines are printed on the lower portion of a page (between the last detail print line and the overflow line) before a new page is started.

You should read [Producing Reports Using LANSA](#) in the *Developers Guide* before attempting to use the DEF_FOOT command.

Portability Considerations Refer to Parameter [TEXT](#).

Also See

[7.20.1 DEF_FOOT Parameters](#)

[7.20.2 DEF_FOOT Comments / Warnings](#)

[7.20.3 DEF_FOOT Examples](#)

Required

```
DEF_FOOT ----- NAME ----- name of footing group ---  
----->
```

Optional

```
>-- FIELDS ----- field name field attributes --->
```

```
      |           |           ||  
      |           --- 7 max ----- |  
      | expandable group expression |  
      ----- 100 max -----
```

```
>-- TEXT ----- 'text' --- line/ --- position -->
```

```
      |           row   column |  
      ----- 50 max -----
```

```
*TMAPnnn 1 1 (special value)
```

```
>-- FOR_REPORT --- 1 ----->
```

report number 1 -> 8

>-- *DESIGN* ----- **ACROSS* ----->
**DOWN*

>-- *IDENTIFY* ----- **LABEL* ----->
**COLHDG*
**NOID*

>-- *DOWN_SEP* ----- *1* ----->
decimal value

>-- *ACROSS_SEP* --- *5* -----|
decimal value

7.20.1 DEF_FOOT Parameters

ACROSS_SEP

DESIGN

DOWN_SEP

FIELDS

FOR_REPORT

IDENTIFY

NAME

TEXT

NAME

Specifies the name that is to be assigned to the group of report print lines defined by this command. The name specified must be unique within the function.

FIELDS

Specifies the field(s) that are to be printed on the report. An expandable group expression is allowed in this parameter.

Only RDML fields are supported.

TEXT

Allows the specification of up to 50 "**text strings**" that are to appear on the screen panel or report. Each text string specified is restricted to a maximum length of 20 characters.

When a text string is specified it should be followed by a row/line number and a column/position number that indicates where it should appear on the screen panel or report.

For example:

```
TEXT(('ACME' 6 2)('ENGINEERING' 7 2))
```

specifies 2 text strings to appear at line 6, position 2 and line 7, position 2 respectively.

Portability

In Visual LANSA this parameter should only be edited using the screen or report painter which will replace any text with a text map. DO NOT enter text using the command prompt or

Considerations

free format editor as it will not pass the full function checker if checked in to LANSAs for i.

All Platforms

The text map is used by the screen or report design facilities to store the details of all the text strings associated with the screen panel or report lines.

Once a screen or report layout has been "painted" and saved, all text details from the layout are stored in a "text map". The text map is then subsequently changed by using the "painter" again.

The presence of a text map is indicated by a TEXT parameter that looks like this example:

```
TEXT((*TMAPnnn 1 1))
```

where "nnn" is a unique number (within this function) that identifies the stored text map.

Some **very important** things about "text maps" and *TMAPnnn identifiers that you **must** know are:

- Never specify *TMAPnnn identifiers of your own or change *TMAPnnn identifiers to other values. Leave the assignment and management of *TMAPnnn identifiers to the screen and report design facilities.
- When copying a command that has an *TMAPnnn identifier, remove the *TMAPnnn references (ie: the whole TEXT parameter) from the copied command. If you fail to do this, then the full function checker will detect the duplicated use of *TMAPnnn identifiers, and issue a fatal error message before any loss occurs.
- Never remove an *TMAPnnn identifier from a command. If this is done then the associated text map may be deleted, or reused in another command, during a full function check or compilation. Loss of text details is likely to result.
- Never "comment out" a command that contains a valid *TMAPnnn identifier. This is just another variation of the preceding warning and it runs the same risks of loss or reuse of text.
- Never specify *TMAPnnn values in an Application Template. In the template context *TMAPnnn values have no meaning. Use the "text string" format in commands used in, and initially generated by, Application Templates.

FOR_REPORT

Specifies the report with which this command should be associated. Up to 8 reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this report is report number 1.

DESIGN

Specifies the design/positioning method which should be used for fields that **do not have specific positioning attributes** associated with them.

*ACROSS, which is the default value for the DEF_FOOT command, indicates that fields should be designed "across" the report line (ie: one after another).

*DOWN indicates that the fields should be designed "down" the report page (ie: one under another).

IDENTIFY

Specifies the default identification method to be used for fields that **do not have specific identification attributes** associated with them.

*LABEL, which is the default value for the DEF_FOOT command, indicates that fields should be identified by their associated labels.

*COLHDG indicates that fields should be identified by their associated column headings.

*NOID indicates that no identification of the field is required. Only the field itself should be included into the report line(s).

DOWN_SEP

Specifies the spacing between lines on the report that should be used when automatically designing a report. The value specified must be a number in the range 1 to 10. The default value for the DEF_FOOT command is 1.

ACROSS_SEP

Specifies the spacing between columns on the report that should be used when automatically designing a report. The value specified must be a number in the range 0 to 10. The default value for the DEF_FOOT command is 5.

7.20.2 DEF_FOOT Comments / Warnings

- When assigning specific line attributes to fields or text in a DEF_FOOT command note that the line numbers used are the actual line numbers that the field will be printed on (unlike the DEF_BREAK and DEF_LINE) commands. Thus specifying *L059 against a field means that the field will always print on report line 59.
- If you use an expandable group expression in a DEF_FOOT command FIELDS parameter and you change the layout using the report design facility, LANSAP will substitute the expression with the actual fields. This is the only way LANSAP can assign attributes to the individual fields, regardless of which group they initially came from.
- Since line numbers specified are the "actual" line numbers, any line numbers specified in a DEF_FOOT command must be after the "last detail print line" and not greater than the "overflow line" number. Refer to the DEF_REPORT command for more information about how these values are derived / set for reports.

7.20.3 DEF FOOT Examples

Refer to [Producing Reports Using LANSAs](#).

7.21 DEF_HEAD

The DEF_HEAD command is used to define one or more heading lines for inclusion on a report.

Only fields of type Alpha, Packed and Signed may be specified. RDMLX field types cannot be specified.

Heading lines are only printed when the condition specified in the TRIGGER_BY parameter is true and are used to begin or trigger a new page in the report.

You should read [Producing Reports Using LANSAs](#) in the *Developers Guide* before attempting to use a DEF_HEAD command.

Portability Considerations Refer to Parameter [TEXT](#) .

Also See

[7.21.1 DEF_HEAD Parameters](#)

[7.21.2 DEF_HEAD Comments / Warnings](#)

[7.21.3 DEF_HEAD Examples](#)

Required

```
DEF_HEAD ----- NAME ----- name of heading group --  
----->
```

Optional

```
>-- FIELDS ----- field name field attributes --->  
|           |           ||  
|           --- 7 max ----- |  
| expandable group expression |  
----- 100 max -----  
  
>-- TRIGGER_BY --- *OVERFLOW -----  
----->  
  
list of field names  
| expandable group expression |  
-----20 max -----
```

```

>-- TEXT ----- 'text' --- line/ --- position -->
      |         row   column |
      ----- 50 max -----
      *TMAPnnn 1 1 (special value)

>-- FOR_REPORT --- 1 ----->
      report number 1 -> 8

>-- DESIGN ----- *ACROSS ----->
      *DOWN

>-- IDENTIFY ----- *LABEL ----->
      *COLHDG
      *NOID

>-- DOWN_SEP ----- 1 ----->
      decimal value

>-- ACROSS_SEP --- 5 -----|
      decimal value

```

7.21.1 DEF_HEAD Parameters

ACROSS_SEP

DESIGN

DOWN_SEP

FIELDS

FOR_REPORT

IDENTIFY

NAME

TEXT

TRIGGER_BY

NAME

Specifies the name that is to be assigned to the group of report print lines defined by this command. The name specified must be unique within the function.

FIELDS

Specifies the field(s) that are to be printed on the report. An expandable group expression is allowed in this parameter.

Only RDML fields are supported.

TRIGGER_BY

Specifies the condition that is to be used to "trigger" the printing of the heading line(s) defined by this command.

*OVERFLOW, which is the default value, indicates that the heading lines should be printed when the previous report page is full (ie: has reached overflow).

Otherwise, specify a list of field names (or an expandable group expression) that are to be used to trigger the printing of the heading line(s). Every time **any** report line is printed LANSa compares the fields nominated in the list with their previous values. If **any field** nominated in the list has changed value the heading line(s) will be produced.

To demonstrate the use of the TRIGGER_BY parameter consider the following example:

```
DEF_HEAD NAME(#HEADING) FIELDS(#COMPANY) TRIGGER_BY(#
```

DEF_LINE NAME(#DETAIL) FIELDS(#REGION #PRODUCT #VALUE)

Note that a new page is triggered by change of #COMPANY, so the report might look like this:

Company : ACME Page 1

Region	Product	Value
XXXXX	XXXXXXXXXXXXXXXXX	999.99
XXXXX	XXXXXXXXXXXXXXXXX	999.99
XXXXX	XXXXXXXXXXXXXXXXX	999.99

Company : ALLIED Page 2

Region	Product	Value
XXXXX	XXXXXXXXXXXXXXXXX	999.99
XXXXX	XXXXXXXXXXXXXXXXX	999.99

Note how page 2 is "triggered" by the change of company name and that page 1 was not full when page 2 was started.

If company ACME had more information than would fit on 1 page, then the report would be produced like this:

Company : ACME Page 1

Region	Product	Value
XXXXX	XXXXXXXXXXXXXXXXX	999.99

Region Product Value Page 2

XXXXX	XXXXXXXXXXXXXXXXX	999.99
XXXXX	XXXXXXXXXXXXXXXXX	999.99
XXXXX	XXXXXXXXXXXXXXXXX	999.99

Region	Product	Value
XXXXX	XXXXXXXXXXXXXXXXX	999.99
XXXXX	XXXXXXXXXXXXXXXXX	999.99

Note how the company name did not print on page 2. This is because the DEF_HEAD command indicates that the heading should only be produced on change of #COMPANY, which did not change on the overflow from page 1 to page 2.

This feature allows the specification of "special" or "break" heading lines. However, in many cases, such as this one, the correct DEF_HEAD command is probably:

```
DEF_HEAD NAME(#HEADING) FIELDS(#COMPANY)
      TRIGGER_BY(#COMPANY *OVERFLOW)
```

This states that the page heading details should be "triggered" by change of #COMPANY **or** page overflow, thus ensuring that the company name prints on every page, even when it is an overflow from the previous page.

TEXT

Allows the specification of up to 50 "**text strings**" that are to appear on the screen panel or report. Each text string specified is restricted to a maximum length of 20 characters.

When a text string is specified it should be followed by a row/line number and a column/position number that indicates where it should appear on the screen panel or report.

For example:

```
TEXT(('ACME' 6 2)('ENGINEERING' 7 2))
```

specifies 2 text strings to appear at line 6, position 2 and line 7, position 2 respectively.

Portability Considerations In Visual LANSA this parameter should only be edited using the screen or report painter which will replace any text with a text map. DO NOT enter text using the command prompt or free format editor as it will not pass the full function checker

if checked in to LANSA for i.

All Platforms

The text map is used by the screen or report design facilities to store the details of all the text strings associated with the screen panel or report lines.

Once a screen or report layout has been "painted" and saved, all text details from the layout are stored in a "text map". The text map is then subsequently changed by using the "painter" again.

The presence of a text map is indicated by a TEXT parameter that looks like this example

```
TEXT((*TMAPnnn 1 1))
```

where "nnn" is a unique number (within this function) that identifies the stored text map.

Some **very important** things about "text maps" and *TMAPnnn identifiers that you **must** know are:

- Never specify *TMAPnnn identifiers of your own or change *TMAPnnn identifiers to other values. Leave the assignment and management of *TMAPnnn identifiers to the screen and report design facilities.
- When copying a command that has an *TMAPnnn identifier, remove the *TMAPnnn references (ie: the whole TEXT parameter) from the copied command. If you fail to do this, then the full function checker will detect the duplicated use of *TMAPnnn identifiers, and issue a fatal error message before any loss occurs.
- Never remove an *TMAPnnn identifier from a command. If this is done then the associated text map may be deleted, or reused in another command, during a full function check or compilation. Loss of text details is likely to result.
- Never "comment out" a command that contains a valid *TMAPnnn identifier. This is just another variation of the preceding warning and it runs the same risks of loss or reuse of text.
- Never specify *TMAPnnn values in an Application Template. In the template context *TMAPnnn values have no meaning. Use the "text string" format in commands used in, and initially generated by, Application Templates.

FOR_REPORT

Specifies the report with which this command should be associated. Up to 8

reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this report is report number 1.

DESIGN

Specifies the design/positioning method which should be used for fields that **do not have specific positioning attributes** associated with them.

*ACROSS, which is the default value for the DEF_HEAD command, indicates that fields should be designed "across" the report line (i.e.: one after another).

*DOWN indicates that the fields should be designed "down" the report page (ie: one under another).

IDENTIFY

Specifies the default identification method to be used for fields that **do not have specific identification attributes** associated with them.

*LABEL, which is the default value for the DEF_HEAD command, indicates that fields should be identified by their associated labels.

*COLHDG indicates that fields should be identified by their associated column headings.

*NOID indicates that no identification of the field is required. Only the field itself should be included into the report line(s).

DOWN_SEP

Specifies the spacing between lines on the report that should be used when automatically designing a report. The value specified must be a number in the range 1 to 10. The default value for the DEF_HEAD command is 1.

ACROSS_SEP

Specifies the spacing between columns on the report that should be used when automatically designing a report. The value specified must be a number in the range 0 to 10. The default value for the DEF_HEAD command is 5.

7.21.2 DEF_HEAD Comments / Warnings

- When assigning specific line attributes to fields or text in a DEF_HEAD command note that the line numbers used are the actual line numbers that the field will be printed on (unlike the DEF_BREAK and DEF_LINE) commands. Thus specifying *L004 against a field means that the field will always print on report line 4.
- If you use an expandable group expression in a DEF_HEAD command FIELDS and/or TRIGGER_BY parameter(s) and you change the layout using the report design facility, LANSAs will substitute the expression with the actual fields. This is the only way LANSAs can assign attributes to the individual fields regardless of which group they initially came from.

7.21.3 DEF_HEAD Examples

Refer to [Producing Reports Using LANSAs](#).

7.22 DEF_LINE

The DEF_LINE command is used to define one or more detail lines for inclusion on a report.

Only fields of type Alpha, Packed and Signed may be specified. RDMLX field types cannot be specified.

A detail line is printed when a PRINT command is executed that nominates it in the LINE parameter.

You should read [Producing Reports Using LANSA](#) in the *Developers Guide* before attempting to use a DEF_LINE command.

Portability Considerations Refer to parameter [TEXT](#).

Also See

[7.22.1 DEF_LINE Parameters](#)

[7.22.2 DEF_LINE Comments / Warnings](#)

[7.22.3 DEF_LINE Examples](#)

Required

```
DEF_LINE ----- NAME ----- name of detail group -----  
----->
```

Optional

```
>-- FIELDS ----- field name field attributes --->  
      |           |           ||  
      |           --- 7 max ----- |  
      | expandable group expression |  
      ----- 1000 max -----
```

```
>-- SPACE_BEF ---- 1 ----->  
      decimal value
```

```
>-- SPACE_AFT ---- 0 ----->  
      decimal value
```

```
>-- TEXT ----- 'text' --- line/ --- position -->
```

| row column |
----- 50 max -----
*TMAPnnn 1 1 (special value)

>-- FOR_REPORT --- 1 ----->
 report number 1 -> 8

>-- DESIGN ----- *ACROSS ----->
 *DOWN

>-- IDENTIFY ----- *COLHDG -----
>
 *LABEL
 *NOID

>-- DOWN_SEP ----- 5 ----->
 decimal value

>-- ACROSS_SEP --- 1 ----->
 decimal value

>-- HEAD_COND ----- *NONE -----
-|
 name of condition

7.22.1 DEF_LINE Parameters

ACROSS_SEP

DESIGN

FIELDS

FOR_REPORT

HEAD_COND

DOWN_SEP

NAME

SPACE_AFT

SPACE_BEF

TEXT

NAME

Specifies the name that is to be assigned to the group of report print lines defined by this command. The name specified must be unique within the function.

FIELDS

Specifies the field(s) that are to be printed on the report. An expandable group expression is allowed in this parameter.

Only RDML fields are supported.

SPACE_BEF

Specifies the number of lines on the report that should be spaced before the detail line(s) are printed. The default value is 1 but any value in the range 0 to 100 can be specified.

SPACE_AFT

Specifies the number of lines on the report that should be spaced after the detail line(s) are printed. The default value is 1 but any value in the range 0 to 100 can be specified.

TEXT

Allows the specification of up to 50 "**text strings**" that are to appear on the screen panel or report. Each text string specified is restricted to a maximum length of 20 characters.

When a text string is specified it should be followed by a row/line number and a column/position number that indicates where it should appear on the screen panel or report.

For example:

```
TEXT(('ACME' 6 2)('ENGINEERING' 7 2))
```

specifies 2 text strings to appear at line 6, position 2 and line 7, position 2 respectively.

Portability Considerations In Visual LANSAs this parameter should only be edited using the screen or report painter which will replace any text with a text map. DO NOT enter text using the command prompt or free format editor as it will not pass the full function checker if checked in to LANSAs for i.

All Platforms

The text map is used by the screen or report design facilities to store the details of all the text strings associated with the screen panel or report lines.

Once a screen or report layout has been "painted" and saved, all text details from the layout are stored in a "text map". The text map is then subsequently changed by using the "painter" again.

The presence of a text map is indicated by a TEXT parameter that looks like this example:

```
TEXT((*TMAPnnn 1 1))
```

where "nnn" is a unique number (within this function) that identifies the stored text map.

Some **very important** things about "text maps" and *TMAPnnn identifiers that you **must** know are:

- Never specify *TMAPnnn identifiers of your own or change *TMAPnnn identifiers to other values. Leave the assignment and management of *TMAPnnn identifiers to the screen and report design facilities.
- When copying a command that has an *TMAPnnn identifier, remove the *TMAPnnn references (ie: the whole TEXT parameter) from the copied command. If you fail to do this, then the full function checker will detect the duplicated use of *TMAPnnn identifiers, and issue a fatal error message before any loss occurs.

- Never remove an *TMAPnnn identifier from a command. If this is done then the associated text map may be deleted, or reused in another command, during a full function check or compilation. Loss of text details is likely to result.
- Never "comment out" a command that contains a valid *TMAPnnn identifier. This is just another variation of the preceding warning and it runs the same risks of loss or reuse of text.
- Never specify *TMAPnnn values in an Application Template. In the template context *TMAPnnn values have no meaning. Use the "text string" format in commands used in, and initially generated by, Application Templates.

FOR_REPORT

Specifies the report with which this command should be associated. Up to 8 reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this report is report number 1.

DESIGN

Specifies the design/positioning method which should be used for fields that **do not have specific positioning attributes** associated with them.

*ACROSS, which is the default value for the DEF_LINE command, indicates that fields should be designed "across" the report line (ie: one after another).

*DOWN indicates that the fields should be designed "down" the report page (ie: one under another).

IDENTIFY

Specifies the default identification method to be used for fields that **do not have specific identification attributes** associated with them.

*COLHDG, which is the default value for the DEF_LINE command, indicates that fields should be identified by their associated column headings.

*LABEL indicates that fields should be identified by their associated labels.

*NOID indicates that no identification of the field is required. Only the field itself should be included into the report line(s).

DOWN_SEP

Specifies the spacing between lines on the report that should be used when automatically designing a report. The value specified must be a number in the range 1 to 10. The default value for the DEF_LINE command is 5.

ACROSS_SEP

Specifies the spacing between columns on the report that should be used when automatically designing a report. The value specified must be a number in the range 0 to 10. The default value for the DEF_BREAK command is 1.

HEAD_COND

Optionally specifies the name of a condition that indicates whether any column heading line(s) associated with fields in **this** detail print line are to be printed in the header area of the report.

*NONE, which is the default value, indicates that no controlling condition applies, and any column headings associated with this detail line should **always** be printed in the report header area, regardless of which detail line is actually being printed.

If a controlling condition is specified, it must be defined elsewhere in the RDML function by a DEF_COND (define condition) command. At the time that **any** print line is to be printed the status of the condition will be checked. Only when it is found to be true will the column headings associated with this detail print line be included in the header area of the report.

7.22.2 DEF_LINE Comments / Warnings

- When assigning specific line attributes to fields or text in a DEF_LINE command note that the line numbers used are "offsets" from the start of the print line. Thus specifying *L001 against a field does not mean the field will actually print on line 1 of the report. The field will print on line 1 of the "group" of fields that make up the DEF_LINE command.
- If you use an expandable group expression in a DEF_LINE command FIELDS parameter and you change the layout using the report design facility, LANSAs will substitute the expression with the actual fields. This is the only way LANSAs can assign attributes to the individual fields, regardless of which group they initially came from.

7.22.3 DEF_LINE Examples

This example applies to the DEF_BREAK command. Refer also to [Producing Reports Using LANSAs](#).

Write an RDML program to read a regional sales file and print details of each record read:

```
DEF_LINE NAME(#DETAIL) FIELDS(#REGION #PRODES #VALUE)
```

```
SELECT FIELDS(#DETAIL) FROM_FILE(SALEHIST)
```

```
PRINT LINE(#DETAIL)
```

```
ENDSELECT
```

```
ENDPRINT
```

7.23 DEF_LIST

The DEF_LIST command is used to define a list and the fields that comprise an entry in the list.

The list may be a **browse** list (used for displaying information at a workstation) or a **working** list (used to store information within a program).

Lists are categorized into static and dynamic lists. A static list is a list that does not specify *MAX in the ENTRYS parameter. If it does it is a dynamic list.

Dynamic lists can only be used in RDMLX objects and are recommended as memory requirements are much lower.

For instance the command:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUAN
```

defines a browse list that can be displayed at a workstation and might look like this:

Order Line Number	Product Number Ordered	Quantity	Net Price
999	XXXXXXXXXX	99999	9999.99
999	XXXXXXXXXX	99999	9999.99
999	XXXXXXXXXX	99999	9999.99
999	XXXXXXXXXX	99999	9999.99
999	XXXXXXXXXX	99999	9999.99
999	XXXXXXXXXX	99999	9999.99

While the command:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUAN
```

defines a working list that can contain at most 10 entries. This type of list cannot be directly displayed at the workstation, but can be accessed within the RDML program just like a browse list.

When the DEF_LIST command is used it defines what fields are in one "entry" of the list. The browse list example above has 6 entries shown and each entry contains an order line number, a product number, a quantity and a price.

Once defined, there are limits to the number of entries in each type of list.

- An RDML browse list can contain up to the number of entries as shown in the

following table:

Operating Systems	RDML	RDMLX List without RDMLX field	RDMLX List with RDMLX field
All (Non Web functions)	9999	9999	9999
IBM i (Web functions)	99999999	9999	99999999
Others (Web functions)	9999	9999	99999999

Continued in [7.23.2 DEF_LIST Description](#)

Portability Considerations Refer to parameter: [SCROLL_TXT](#).

Also See

[7.23.1 DEF_LIST Parameters](#)

[7.23.2 DEF_LIST Description](#) continued

[7.23.3 DEF_LIST Comments / Warnings](#)

[7.23.4 DEF_LIST Examples](#)

```

                                Required
DEF_LIST ----- NAME ----- name of list -----
->
    >-- FIELDS ----- field name field attributes --->
        |           |           ||
        |           --- 7 max ----- |
        | expandable group expression |
        |----- 1000 max for RDMLX----|
        |----- 100 max for RDML -----
-----
                                Optional
    >-- COUNTER ----- *NONE -----
>

```

numeric field name

>-- TYPE ----- *BROWSE ----->
 *WORKING

>-- ENTRYS ----- 50 ----->
 number in range 1 – 2,147,483,647

>-- PAGE_SIZE ---- *NONE ----->
 numeric field name

>-- TOP_ENTRY ---- *CURRENT -----

-->

numeric field name

>-- SEL_ENTRY ---- *NONE -----

>

numeric field name

>-- SCROLL_TXT --- *NONE -----

|

alphanumeric field name

7.23.1 DEF_LIST Parameters

COUNTER

ENTRYS

FIELDS

NAME

PAGE_SIZE

SCROLL_TXT

SEL_ENTRY

TOP_ENTRY

TYPE

NAME

Specifies the name that is to be assigned to the list. The name specified must not be the same as any other list defined in this function, any group name defined in this function (see the GROUP_BY command), or any field defined in this function or in the LANSAs data dictionary.

FIELDS

Specifies the names of the field(s) which are to form one entry in the list. Field control attributes can be included with the field(s) specified. Alternatively, an expandable group expression can be entered in this parameter.

All fields nominated in this parameter must be defined within this function or in the LANSAs data dictionary.

Fields of type Binary, VarBinary, Float and Boolean cannot be used.

Refer to the [7.52.1 GROUP_BY Parameters](#) for more details of specifying field lists and field control attributes.

COUNTER

Specifies a field which is to be automatically maintained with a count of the number of entries currently in the list.

The default value is *NONE which indicates no automatic counter is required.

If a field is nominated as the automatic counter it must be defined in this function or in the LANSAs data dictionary and be of numeric type.

TYPE

Identifies the list as either a browse list (*BROWSE) which can be displayed at a workstation, or as a working list (*WORKING) that can be used within the program. The default value for this parameter is *BROWSE.

ENTRYS

This parameter is used for TYPE(*WORKING) lists or for TYPE(*BROWSE) lists when the application will be run on Visual LANSA. It specifies the maximum number of entries that can be contained in the list at any time. If this parameter is omitted a value of 50 is assumed. Otherwise specify an integer value in the range 1 to 2,147,483,647. If the Function or Component is not enabled for RDMLX, the limit is 9999.

The special value *MAX is provided to represent 2,147,483,647. Using this value on some operating systems, like 32-bit Microsoft Windows versions, will overflow available memory in a process if an attempt is made to add the maximum number of entries. So, the use of a large number of entries needs to be seen as allowing for an unknown number that may never be practically attained. It is up to the application designer to make appropriate use of this feature. The full function checker issues warnings when the maximum size of the list would overflow the practical available memory on various MS Windows versions. Refer to [7.23.3 DEF_LIST Comments / Warnings](#) for information on how to use the FFC warnings to calculate memory use.

Note that an RDML list pre-allocates all the required memory for the list. An RDMLX list pre-allocates only the number of entries that will fit into one memory page or the width of one entry, whichever is the larger. RDMLX lists allocate more memory at runtime as the need arises.

PAGE_SIZE

This parameter is only used for TYPE(*BROWSE) lists. If used, it specifies the name of a numeric field that is to contain the size (number of entries) of the browse list that fits on one "page" of the display device being used.

The default value is *NONE which indicates that page size is not required.

If a field is nominated to contain the "page" size it must be defined in this function or in the LANSA data dictionary and be of numeric type.

Note: The use of this parameter **does not control** the page size, it merely allows you to nominate a numeric field that is to have the LANSA assigned page size placed into it at the start of the RDML program.

This parameter is normally only used when creating browse lists that are displayed using "page at a time" techniques. It saves the programmer from having to find out what the LANSAs assigned page size is, and then having to "hard code" it into the RDML program.

TOP_ENTRY

This parameter is only used for TYPE(*BROWSE) lists. It specifies a numeric field which can be used to:

- Specify that the "page" of the browse list that is to be displayed initially is the "page" that contains this entry number.
- Receive back (after a DISPLAY, REQUEST or POP_UP command) the browse list entry number of the browse list line that was at the top of the "page" when the enter key or a function key was used.

The default value is *CURRENT which indicates that the "page" to be displayed should be the current one. This means the first page if this is the first time the list is being displayed, or the one that was last displayed if the list has been previously displayed.

If a field is nominated to contain the top entry it must be defined in this function or in the LANSAs data dictionary and be of numeric type.

SEL_ENTRY

This parameter is only used for TYPE(*BROWSE) lists. It specifies a numeric field which is to contain the entry number of a browse list entry that was selected by the user placing the cursor on the entry and pressing enter or a function key. If the cursor is not validly positioned the value will be returned as zero.

The default value is *NONE which indicates that cursor selected entry number is not required.

If a field is nominated to contain the selected entry number it must be defined in this function or in the LANSAs data dictionary and be of numeric type.

SCROLL_TXT

This parameter is only used for TYPE(*BROWSE) lists. Additionally it should **only be used** when you are coding your RDML program to use "**page at a time**" browse list displays.

Under the current releases of LANSAs, IBM i and CPF this parameter is used to control the appearance (or non-appearance) of the high intensity "+" (plus) sign at the bottom of a list displayed on the screen.

Traditionally, this appearance of the "+" sign indicates to the user that more details exist on the next "page" of the list, and that they can be viewed by using the roll up key.

When your RDML program is **not using "page at a time" techniques** to build and display a list (ie: you build the whole list before displaying it) you should **not use** this parameter.

If your RDML program is **using "page at a time" techniques**, you should use this parameter to control when the "+" scrolling sign appears.

The default value is *NONE, which indicates that no specific program control of the "+" sign is required. It will be handled automatically by LANSAs, IBM i or CPF with no program intervention. Use this value when you are not using "page at a time" techniques.

Otherwise nominate an alphanumeric field name for this parameter. When **any** page from the browse list is displayed on the screen, either by the RDML program or by the user pressing one of the scroll keys to move backwards or forwards through the list, the following occurs:

- If **more entries** exist in the browse list beyond the page that is being displayed, the "+" sign will appear, regardless of what value is contained in the field you nominate in this parameter.
- If **no more entries** exist in the browse list beyond the page that is being displayed, the appearance of the "+" sign is controlled by the content of the field you nominate in this parameter. If the contents of the field nominated **start with** an "M" or an "m", the "+" sign will appear. Otherwise it will not appear.

Refer to the end of this section for an example of a "page at a time" browse list program and the use of this parameter.

Portability Considerations

If used with Visual LANSAs, this feature is ignored.

7.23.2 DEF_LIST Description

continued from [7.23 DEF_LIST](#)

- An RDMLX browse list can only be used on the Web and in this context list use greatly impacts the response time in the browser. Client and server computing power and the size of the communication pipe will dictate what the practical limit is. In some configurations it can be as little as 1000 entries. Notice that a browse list is a static list. A static list allocates sufficient tracking information for the maximum number of entries specified. For small numbers of entries, such as 10,000, this tracking information is inconsequential. But, if millions of entries are required, it can become significant.
- Because of a limitation of generated field names for an HTML form, only 9999 entries can be used for input. Due of this limitation, for an RDMLX/RDML browse list used on the Web and which supports more than 9999 entries, the entries beyond 9999 are for output only.
- An RDMLX function can only use a browse list if it is web enabled.
- A static working list in an RDML object can contain up to the number of entries specified in the ENTRYS parameter which has a maximum of 9999. However, the aggregate entry length cannot exceed 256 bytes in a primary list. See the note in [7.23.3 DEF_LIST Comments / Warnings](#).
- A static working list in an RDMLX object can also contain up to the number of entries specified in the ENTRYS parameter which has a maximum of 2 giga entries. The aggregate entry length cannot exceed 2 Giga bytes in a primary list. Also, String and Binary data memory needs are on top of this as they are not stored in the list itself. Thus each entry could have many Strings each up to 64 Kbytes long. It is very easy to consume very large amounts of memory. See the note in [7.23.3 DEF_LIST Comments / Warnings](#).
- A dynamic working list in an RDMLX object allocates and releases memory on demand. Enter the value *MAX into the ENTRYS parameter. This is the kind of list recommended for use in an RDMLX object, though it has severe restrictions when used with the SORT_LIST command. The aggregate entry length cannot exceed 2 Giga bytes in a primary list. Also, String and Binary data memory needs are on top of this as they are not stored in the list itself. Thus each entry could have many Strings each up to 64 Kbytes long. It is very easy to consume very large amounts of memory, far beyond the capacities of today's computers. The memory management is described in

7.23.3 DEF_LIST Comments / Warnings.

The actual positioning of a browse list onto the workstation display depends upon the parameters used in the REQUEST, DISPLAY or POP_UP command that is used to display the list on the screen and on any field attributes used in the DEF_LIST command. For more details, refer to [Field Attributes and their Use](#).

When a browse list is displayed at a workstation only the first "page" is displayed. A "page" is the number of entries that will fit on the screen. By using the ROLL UP and ROLL DOWN keys the user can browse backwards and forwards through all the pages in the list. This is why it is called a "browse" list. Generally a browse list should only be used when the list entries are to be displayed at a workstation. Working lists, which cannot be directly displayed at a workstation, have 2 major advantages over browse lists. The first is that they can be processed much faster than browse lists, and the second is that they can be used in RDML programs running in batch.

Some of the other commands that work with or reference lists include:

Command	Description	Valid For Browse List	Valid For Working List
ADD_ENTRY	Add a new entry to a list.	YES	YES
UPD_ENTRY	Update an existing entry in a list	YES	YES
GET_ENTRY	Get an entry from a list	YES	YES
SELECTLIST	Process entries from a list in a loop	YES	YES
CLR_LIST	Clear all entries from a list	YES	YES
DLT_LIST	Delete a list	YES	YES
INZ_LIST	Initialise a list with "n" entries	YES	YES
DISPLAY	Display fields and optionally a list	YES	NO
REQUEST	Request fields and optionally a list	YES	NO
POP_UP	Display fields and optionally a list in a pop up window	YES	NO

LOC_ENTRY	Locate an entry in a list	NO	YES
SORT_LIST	Sort a list	NO	YES
DLT_ENTRY	Delete entry from a list	NO	YES

7.23.3 DEF_LIST Comments / Warnings

- There are two types of working list. The first list is one that specifies *MAX for the ENTRYS parameter. This kind of list dynamically allocates memory and is referred to as a Dynamic Working List. The second specifies any other value for the ENTRYS parameter. These are static working lists.
- **Warnings**
 - Keep in mind when defining static working lists that the amount of storage allocated to each working list will be equal to the entry length multiplied by the number of entries on the list, so the amount of storage space allocated to a function using many working lists can increase quite substantially.
 - If you intend to improve the memory management and performance of your lists by performing storage space manipulation, a document: *Memory Management in LANSAs* that covers this topic and is available from your LANSAs distributor.
Under normal circumstances you should not need to change a list's storage space. Changing it without understanding the implications could affect your application's performance or stability. Contact your local LANSAs distributor before continuing.
- When a static working list will exceed typical available Windows 32-bit process memory, messages 870, 871, 872, 873 and 874 may be displayed. An example of message 871 is: "Maximum 32-bit windows server process memory of 3 GB will be exceeded." These messages should be considered as near fatal errors. They are indicating that the memory requirement is beyond the capability of particular windows configurations. The capabilities of other platforms is generally larger, like the IBM i, but to raise these warnings still indicates a design that should be re-considered.
- Message 874 contains the dimensions of the list. An example of message 874 is "List page size = 1098000000 bytes Entry length = 549 bytes."
- On the other hand a dynamic working list only pre-allocates a small amount of memory to hold pointers to the list entries. Then, as more space is required it is allocated with one page of operating system memory or the size of one entry, whichever is the larger. On Microsoft Windows the size of a page is 32 KB. Memory is also released as entries are deleted from the list. If you were to keep adding entries indefinitely, memory would eventually run out on windows.

- Whilst the IBM i has a greater total amount of memory available, it is limited to a maximum of 16 MB in any single memory allocation. This means that on an IBM i, each **STATIC** working list is limited to a TOTAL size of 16 MB and each **DYNAMIC** working lists is limited to a maximum ENTRY width of 16 MB. Thus a dynamic working list has a far greater total capacity - only limited by the total amount of memory that the operating system has available for the process to use. This is strictly an IBM i limitation. All other platforms have the same limit for each list as for the total memory used by all lists.
- Large working lists may not perform satisfactorily, especially if LOC_ENTRY is used because they are only accessed sequentially - there are no indexes. For lists that allow fast look-up see the SPACE BIFs.

Note: Once defined, an RDML working list has an aggregate length limit of 256 bytes in the primary list. To overcome this limit Appendage Lists may be specified by invoking the [Transform_LIST](#) Built-In Function. Another solution is to use an RDMLX Function with a dynamic working list.

Working lists are not an infinite resource - use a realistic and sensible size for the number of entries.

- Where the fields defined in a browse list will not fit onto one screen line they will automatically wrap around onto the next screen line. For example, the command:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
```

May cause a list to be displayed that looks like this on the screen:

No	Product	Quantity	Price
99	9999999	99999	99999.99
99	9999999	99999	99999.99
99	9999999	99999	99999.99

If more fields were added to the DEF_LIST command like this:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #PDE
```

then when the list was displayed it might now look like this:

No	Product	Description
-----------	----------------	--------------------

```

99  9999999          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Quantity 99999      Price 99999.99   Tax rate 99.99
99  9999999          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Quantity 99999      Price 99999.99   Tax rate 99.99
99  9999999          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Quantity 99999      Price 99999.99   Tax rate 99.99

```

Field attributes and/or the screen design facility can be used to modify the automatically designed screen layout in any way.

For information refer to:

In this guide, for details of:

- field attributes, refer to [Field Attributes](#).
- the screen design facility, refer to The Screen Design Facility in the *Visual LANSA User Guide*. Note that row/line attributes are ignored when specified for fields in a browselist.

In the *LANSA for i User Guide*, for details of:

- field attributes, refer to New Field Attribute Concepts [Create a New Field Definition](#) .
- the screen design facility, refer to [The Screen Design Facility](#). Note that row/line attributes are ignored when specified for fields in a browselist.

7.23.4 DEF_LIST Examples

Example 1: Write an RDML program to display the full details of an order. Define the order line list required with name #ORDERLINE and group the fields required from the order header file under the name #ORDERHEAD:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
GROUP_BY NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #
```

Ask the user to input an order number, clear all entries from the list and set mode to display:

```
L1: REQUEST FIELDS(#ORDNUM)
      SET_MODE TO(*DISPLAY)
      CLR_LIST NAMED(#ORDERLINE)
```

Fetch the required fields from the ORDHDR file. If not found return to REQUEST command with an automatic error message:

```
FETCH FIELDS(#ORDERHEAD) FROM_FILE(ORDHDR) WITH_KEY
```

Select the required fields from the ORDLIN file. For each record selected add a new entry to the list named #ORDERLINE:

```
SELECT FIELDS(#ORDERLINE) FROM_FILE(ORDLIN) WITH_KEY(#
ADD_ENTRY TO_LIST(#ORDERLINE)
ENDSELECT
```

Finally display the order header fields and order line details to the user:

```
DISPLAY FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)
```

The screen formats automatically designed by LANSAs for this RDML program would look something like this:

For the **REQUEST FIELDS(#ORDNUM)** command:

Order number : _____

For **DISPLAY FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)** command:

Order number : 99999999
Customer no : 999999
Date due : 99/99/99

Line
No Product Quantity Price
99 9999999 99999 99999.99
99 9999999 99999 99999.99
99 9999999 99999 99999.99
99 9999999 99999 99999.99
99 9999999 99999 99999.99
99 9999999 99999 99999.99

Example 2: Modify the RDML program used in the previous example to include the field #PDESC (product description) from the PROMST (product master) file into the list:

Include #PDESC into the list definition.

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #PDESC  
GROUP_BY NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #
```

```
L1: REQUEST FIELDS(#ORDNUM)  
SET_MODE TO(*DISPLAY)  
CLR_LIST NAMED(#ORDERLINE)  
FETCH FIELDS(#ORDERHEAD) FROM_FILE(ORDHDR) WITH_KE
```

Select the required fields from the ORDLIN file. For each record selected get the associated product description then add a new entry to the list named #ORDERLINE:

```
SELECT FIELDS(#ORDERLINE) FROM_FILE(ORDLIN) WITH_KEY(#  
FETCH FIELDS(#PDESC) FROM_FILE(PROMST) WITH_KEY(#PROD  
ADD_ENTRY TO_LIST(#ORDERLINE)  
ENDSELECT
```

```
DISPLAY FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)
```

The screen formats automatically designed by LANSA for this amended RDML

program would now look something like this:

For the REQUEST FIELDS(#ORDNUM) command:

Order number : _____

For DISPLAY FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE) command:

Order number : 99999999

Customer no : 999999

Date due : 99/99/99

Line

No	Product Description	Quantity	Price
99	XXXXXXXXXXXXXXXXXXXXXXXXXX	99999	99999.99
99	XXXXXXXXXXXXXXXXXXXXXXXXXX	99999	99999.99
99	XXXXXXXXXXXXXXXXXXXXXXXXXX	99999	99999.99
99	XXXXXXXXXXXXXXXXXXXXXXXXXX	99999	99999.99
99	XXXXXXXXXXXXXXXXXXXXXXXXXX	99999	99999.99
99	XXXXXXXXXXXXXXXXXXXXXXXXXX	99999	99999.99

Example 3: Consider the following simple RDML program that requests the user inputs the surname (fully or partially), then displays a list of all employees whose names start with the value specified:

```
***** Define work variables and browse list to be used
DEFINE FIELD(#L1COUNT) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEF_LIST NAME(#L1) FIELDS((#SURNAME) (#GIVENAME) (#EMPNC
***** Loop until terminated by EXIT or CANCEL
BEGIN_LOOP
***** Get surname to search for
REQUEST FIELDS(#SURNAME)
***** Build list of generically identical names
CLR_LIST NAMED(#L1)
SELECT FIELDS(#L1) FROM_FILE(PSLMST2) WITH_KEY(#SURNAM
ADD_ENTRY TO_LIST(#L1)
ENDSELECT
***** If names found, display list to user
```

```

IF      COND('#L1COUNT *GT 0')
DISPLAY BROWSELIST(#L1)
***** else issue error indicating none found
ELSE
MESSAGE MSGTXT('No employees have a surname matching request')
ENDIF
***** Loop back and request next name to search for
END_LOOP

```

This program will work just fine, but what if the user inputs a search name of "D", and 800 employees working for the company have a surname that starts with "D"?

The result will be a list containing 800 names. But more importantly, it will take a **long time** to build up the list and use a **lot of computer resource** while doing it.

To solve this problem, a technique called "page at a time" browsing is often used. What this basically means is that the program extracts one "page" of names matching the request, and then displays them to the user. If the user presses the roll up key then the next page is fetched and displayed, etc, etc.

To implement a "page at a time" technique for this particular program it could be modified like this:

```

***** Define work variables and browse list to be used
DEFINE  FIELD(#L1COUNT) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE  FIELD(#L1PAGE) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE  FIELD(#L1TOP) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE  FIELD(#L1POS) TYPE(*CHAR) LENGTH(7)
DEF_LIST NAME(#L1) FIELDS((#SURNAME) (#GIVENAME) (#EMP
***** Loop until terminated by EXIT or CANCEL
BEGIN_LOOP
***** Get surname to search for
REQUEST  FIELDS(#SURNAME)
***** Build list of generically identical names
CLR_LIST NAMED(#L1)
CHANGE  FIELD(#IO$KEY) TO(UP)
CHANGE  FIELD(#L1TOP) TO(1)
SELECT  FIELDS(#L1) FROM_FILE(PSLMST2) WITH_KEY(#SURN.
EXECUTE  SUBROUTINE(DISPLAY) WITH_PARMS(''More...'')
ADD_ENTRY TO_LIST(#L1)

```

```

ENDSELECT
***** If names found, display list to user
IF COND('#L1COUNT *GT 0')
EXECUTE SUBROUTINE(DISPLAY) WITH_PARMS(''Bottom'')
***** else issue error indicating none found
ELSE
MESSAGE MSGTXT('No employees have a surname matching request')
ENDIF
***** Loop back and request next name to search for
END_LOOP
*****
***** Display names if page is full or list is complete
*****
SUBROUTINE NAME(DISPLAY) PARMS(#L1POS)
DEFINE FIELD(#L1REMN) TYPE(*DEC) LENGTH(5) DECIMALS(0)
CHANGE FIELD(#L1REMN) TO('#L1COUNT / #L1PAGE')
IF COND('#L1COUNT *NE 0') *AND (#IO$KEY = UP) *AND ((#L1
DUNTIL COND('#L1POS *NE 'Bottom') *OR (#IO$KEY *NE UP)
DISPLAY BROWSELIST(#L1) USER_KEYS((*ROLLUP))
ENDUNTIL
CHANGE FIELD(#L1TOP) TO('#L1TOP + #L1PAGE')
ENDIF
ENDROUTINE

```

The "page at a time" technique described here can be applied to just about any situation where a browse list is to be displayed and can considerably improve performance in most of them.

It is easy to modify existing programs that use SELECT and DISPLAY (like the initial example here) to use the page at a time technique. Note how the new logic "slots into" the existing logic with no major structural change to the program logic/flow.

The easiest way to implement "page at a time" techniques is to design and fully test a standard "algorithm" that is suitable for your site's needs. This can then be used as a base or template for all future applications.

Example 4: A transaction file called TRANS is to be printed and contains 10,000 records. For each transaction printed the associated state description must be extracted from file STATES and printed as well.

A simple RDML program to do this might look like this:

```

GROUP_BY  NAME(#TRANS) FIELDS(#TRANNUM #TRANTIME #TRA
SELECT  FIELDS(#TRANS) FROM_FILE(TRANS)
FETCH  FIELDS(#TRANS) FROM_FILE(STATES) WITH_KEY(#TRANS
UPRINT  FIELDS(#TRANS)
ENDSELECT

```

However, by using a working list the speed of this program can be improved considerably:

```

GROUP_BY  NAME(#TRANS) FIELDS(#TRANNUM #TRANTIME #TRA
DEF_LIST  NAME(#STATES) FIELDS(#STATE #STATEDES) TYPE(*WOI

```

```

SELECT  FIELDS(#STATES) FROM_FILE(STATE)
ADD_ENTRY TO_LIST(#STATES)
ENDSELECT

```

```

SELECT  FIELDS(#TRANS) FROM_FILE(TRANS)
LOC_ENTRY IN_LIST(#STATES) WHERE('#STATE = #TRANSTATE')
UPRINT  FIELDS(#TRANS)
ENDSELECT

```

If there were 10 states, then this version of the print program would do 9,990 less database accesses than the first version.

Note that exactly the same performance improvement can be achieved more simply by using the KEEP_LAST parameter of the FETCH command like this:

```

GROUP_BY  NAME(#TRANS) FIELDS(#TRANNUM #TRANTIME #TRA
SELECT  FIELDS(#TRANS) FROM_FILE(TRANS)
FETCH  FIELDS(#TRANS) FROM_FILE(STATES) WITH_KEY(#TRANS
UPRINT  FIELDS(#TRANS)
ENDSELECT

```

7.24 DEF_REPORT

The DEF_REPORT command is used to define the attributes of a report.

A DEF_REPORT command is only required in a function when the attributes of the report are different from the default report attributes set up within your LANSAs system.

Before you use DEF_REPORT, refer to [Producing Reports Using LANSAs](#) for further information.

Also See

[7.24.1 DEF_REPORT Parameters](#)

[7.24.2 DEF_REPORT Comments / Warnings](#)

[7.24.3 DEF_REPORT Examples](#)

Optional

```
DEF_REPORT --- REPORT_NUM --- 1 -----
----->
      report number 1 -> 8

>-- PRT_FILE ----- QSYSPRT ----->
      printer file.library name

>-- FORMSIZE ---- *DEFAULT ---- *DEFAULT ---
----->
      *FILE ---- *FILE
      form length form width

>-- LAST_LINE ---- *DEFAULT -----
>
      *FILE
      last print line number

>-- OVERFLOW ----- *DEFAULT -----
-->
      *FILE
      overflow line number
```

>-- *OUTQ* ----- **FILE* ----->
 output queue name

>-- *COPIES* ----- **FILE* ----->
 number of copies

>-- *FORMTYPE* ----- **FILE* ----->
 type of forms

>-- *OTHER_OVR* ---- **NONE* -----
->
 other override attributes

>-- *RET_LENGTH* --- **NONE* -----
->
 returned form length

>-- *RET_WIDTH* ---- **NONE* -----
>
 returned form width

>-- *RET_OVERF* ---- **NONE* -----
>
 returned overflow value

>-- *RET_LINE* ----- **NONE* ----->
 returned line number

>-- *RET_PAGE* ----- **NONE* -----|
 returned page number

7.24.1 DEF_REPORT Parameters

COPIES

FORMSIZE

FORMTYPE

LAST_LINE

OTHER_OVR

OUTQ

OVERFLOW

PRT_FILE

REPORT_NUM

RET_LENGTH

RET_LINE

RET_OVERF

RET_PAGE

RET_WIDTH

REPORT_NUM

Specifies the report which is to be defined by this command. Up to 8 reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this report is report number 1.

PRT_FILE

IBM i use only

Specifies the name and library of residence of the IBM i printer file that should be used to produce this report.

QSYSPRT.*LIBL, which is the default value, indicates that the IBM supplied default printer file QSYSPRT (which is to be located by using the job's library list) should be used to produce this report.

Optionally, specify the name of an IBM i printer file and either *LIBL or the library in which the file resides. Printer files must be created using CRTPRTF command and are used to describe various attributes of the printer file. Refer to the appropriate IBM manual for details of the CRTPRTF command.

Printer files other than QSYSPRT are usually only required when some "special" printer file attributes are required for the report. An example might be

when the lines per inch, characters per inch or printer quality attributes are different to those in the IBM supplied default printer file QSYSPRT.

When a printer file is specified, spool file User Data must be specified in the printer file or, for RDMLX, the OTHER_OVR parameter.

FORMSIZE

Specifies the length and width of the form that the report is to be produced on.

Not supported in RDMLX.

*DEFAULT/*DEFAULT, which are the default values, indicate that the formsize parameters should be extracted from the LANSAs system definition block. Refer to [The System Definition Data Areas](#) in the *LANSAs for i User Guide* for details of the system definition block and how it can be changed. In the shipped LANSAs system definition block the form length is set at 66 and the form width at 132.

*FILE/*FILE, indicate that the form size parameters should be extracted from the values specified on the printer file. These values are only taken at execution time.

Otherwise nominate the length and width of the form on which the report is to be produced.

LAST_LINE

Specifies the last line upon which report detail lines (DEF_LINE and DEF_BREAK lines) should be allowed to be printed.

Not supported in RDMLX.

*DEFAULT, which is the default value, indicates that the value used should be calculated as 3 less than the overflow line specified in the system definition block. In the shipped version of LANSAs the overflow line number is set at 60, so the default last print line number is 57.

*FILE, indicates that the value used should be calculated as 3 less than the overflow line specified on the printer file.

Otherwise, specify the last print line number required.

OVERFLOW

Specifies overflow line number to be used for the report. No print line (no matter what type) can be printed beyond the overflow line on a report.

Not supported in RDMLX.

*DEFAULT, which is the default value, indicates that the overflow parameters

should be extracted from the LANSAs system definition block. Refer to [The System Definition Data Areas](#) in the *LANSAs for i User Guide* and how it can be changed. In the shipped LANSAs system definition block the overflow line number is set to 60.

*FILE, indicates that the overflow parameters should be extracted from the printer file.

Otherwise nominate overflow line number required.

Note: Report footlines can only be defined on lines beyond the last print line and up to and including the overflow line. Thus if LANSAs shipped values are used, report footlines can only be defined on lines 58 to 60.

OUTQ

Specifies the output queue name for the spooled output file. Note that library name cannot be specified for the OUTQ parameter.

*FILE indicates that the output queue name should be taken from the default output queue name for the printer file.

COPIES

Specifies the number of copies that are to be produced for the report. A value from 1 to 255 is allowed.

*FILE indicates that the number of copies should be taken from the default value for the printer file.

FORMTYPE

Specifies the type of forms used in the printer for the report. If a form type other than the default is used, the system (when the output is produced) sends a message that identifies the form type to the system operator and requests that the specified type of forms be put in the printer.

*FILE indicates that the form type should be taken from the default value for the printer file.

OTHER_OVR

Additional IBM i printer file override options can be specified in this parameter, for example HOLD(*YES). Refer to the IBM Control Language Reference Manuals for additional overrides. The following overrides cannot be specified in this parameter as they are already handled within the DEF_REPORT command:

FILE LVLCHK - (specified as *NO)
TOFILE SECURE - (specified as *NO)
OVRFLOW SHARE - (specified as *NO)
OUTQ PAGESIZE - (on the IBM i)
FORMTYPE COPIES

*NONE indicates that there are no additional overrides.

Note - this parameter allows entry of additional printer file overrides which rely heavily on IBM i features - this must be taken into consideration when future use on other platforms may be likely.

RET_LENGTH

Specifies whether the form length should be returned into the field named in the parameter. The field must be defined as a numeric field.

*NONE indicates that the form length will not be returned.

RET_WIDTH

Specifies whether the form width should be returned into the field named in the parameter. The field must be defined as a numeric field.

*NONE indicates that the form width will not be returned.

RET_OVERF

Specifies whether the overflow value for the printer file should be returned into the field named in the parameter. The field must be defined as a numeric field.

*NONE indicates that the overflow value will not be returned.

RET_LINE

Specifies whether the current line number for the printer file should be returned into the field named in the parameter. The field must be defined as a numeric field.

*NONE indicates that the current line number will not be returned.

RET_PAGE

Specifies whether the current page number should be returned into the field named in the parameter. The field must be defined as a numeric field.

Note that the page number is always returned after the first line of a page is

printed. This means that the page number in RET_PAGE is always 1 page behind the printed page.

*NONE indicates that the current page number will not be returned.

7.24.2 DEF_REPORT Comments / Warnings

- When using value *FILE for form length, last line and overflow, make the use of *FILE consistent, that is do not for example use *DEFAULT for form length while using *FILE on overflow unless you are sure the two values will not conflict.

Portability Note:

- PRT_FILE parameter is for use with IBM i only.
- FORMSIZE, LAST_LINE & OVERFLOW parameters are not supported in RDMLX.

7.24.3 DEF_REPORT Examples

Define report number 1 to be printed on a 198 character printer

Define report number 3 to be printed via print file INVOICE

Define report number 1 to have 3 copies printed on output queue LASER

Define report number 1 to have form length returned in field #RETLEN

Refer also to [Producing Reports Using LANSA](#).

Define report number 1 to be printed on a 198 character printer

```
DEF_REPORT FORMSIZE(66 198)
```

Define report number 3 to be printed via print file INVOICE

INVOICE has a length of 50 and a width of 80. Last print line and overflow line are set to 48 and 49 respectively:

```
DEF_REPORT REPORT_NUM(3) PRT_FILE(INVOICE) FORMSIZE(50 80
```

Define report number 1 to have 3 copies printed on output queue LASER

Use a formtype of A4 LETTER and for the output to be held:

```
DEF_REPORT OUTQ(LASER) COPIES(3) FORMTYPE('A4 LETTER') O'
```

Define report number 1 to have form length returned in field #RETLEN

Form width is returned in field #RETWID, the overflow value is returned in field #RETOVF, current line number is returned in field #RETLIN and current page number is returned in field #RETPAG.

```
DEF_REPORT RET_LENGTH(#RETLEN) RET_WIDTH(#RETWID) RET'
```

7.25 DEFINE

The DEFINE command is used to define a field for local use within a function, form, reusable part or WAM.

Also See

[7.25.1 DEFINE Parameters](#)

[7.25.2 DEFINE Examples](#)

Required

```
DEFINE ----- FIELD ----- field name -----  
>
```

Optional

```
>-- TYPE ----- *REFFLD ----->  
    *DEC  
    *PACKED  
    *CHAR  
    *NVARCHAR  
    *STRING  
    *SIGNED  
    *BIN  
    *DATE  
    *TIME  
    *DATETIME  
    *INT  
    *FLOAT  
    *BOOLEAN  
>-- LENGTH ----- *REFFLD -----  
>  
    numeric value  
    incr/decr    *PLUS  
                *MINUS  
                *NONE  
    # to incr/decr *NONE  
                numeric value  
>-- DECIMALS ----- *REFFLD -----
```

```

>
    numeric value
    incr/decr  *PLUS
               *MINUS
               *NONE
    # to incr/decr *NONE
    numeric value
>-- REFFLD ----- *NONE ----->
    name of reference field
>-- LABEL ----- *DEFAULT ----->
    label name

>-- DESC ----- *DEFAULT ----->
    text description

>-- COLHDG ----- *DEFAULT -----
->
    column heading
    |         |
    - 3 maximum -

>-- EDIT_CODE ---- *DEFAULT -----
-->
    edit code

>-- EDIT_WORD ---- *DEFAULT -----
-->
    edit word

>-- INPUT_ATR ---- *DEFAULT -----
->
    input attributes

>-- OUTPUT_ATR --- *DEFAULT -----
--->
    output attributes

>-- DEFAULT ----- *DEFAULT -----
>

```

default value

-> >-- *TO_OVERLAY* --- **NONE* ----- 1 -----

#field name start position

>-- *SHIFT* ----- **DEFAULT* -----|
keyboard shift

7.25.1 DEFINE Parameters

COLHDG

DECIMALS

DEFAULT

DESC

EDIT_CODE

EDIT_WORD

FIELD

INPUT_ATR

LABEL

LENGTH

OUTPUT_ATR

REFFLD

SHIFT

TO_OVERLAY

TYPE

FIELD

Specifies the name of the field which is to be defined. The field name must start with a # and not be defined in the LANSAs data dictionary. In addition, it must not be the name of a group or list defined within this function. Avoid the use of field names like SQLxxx, as this may cause problems when used in functions that use SQL (Structured Query Language) facilities. (IE Command SELECT_SQL.)

TYPE

Specifies the type of field which is being defined. The permissible values are:

- *BIN indicates a Binary working field.
- *BOOLEAN indicates a Boolean working field. Valid values for use with a Boolean are True and False (not case sensitive) or 1 and 0 (zero).
- *CHAR indicates an Alpha working field.
- *DATE indicates a Date working field.
- *DATETIME indicates a Datetime working field.

- *DEC or synonym *PACKED indicates a Packed working field.
- *FLOAT indicates a Float working field.
- *INT indicates an Integer working field.
- *NVARCHAR indicates an NVarChar working field.
- *REFFLD indicates the type comes from the Reference field in the REFFLD parameter.
- *SIGNED indicates a Signed working field.
- *STRING indicates a String working field.
- *TIME indicates a Time working field.

If the Function or Component is not RDMLX enabled, only *REFFLD, *DEC, and *CHAR are valid types.

If the REFFLD parameter is not *NONE:

- and the referenced field is an RDMLX field
the TYPE parameter MUST be specified as *REFFLD.
- and the reference field is an RDML field
the TYPE parameter may only be set to *REFFLD, *DEC or *CHAR.
Specifying the type as other than *REFFLD is not recommended as it renders useless the fundamental idea of repository fields.

LENGTH

Specifies the length of the field being defined. If the value *REFFLD is specified then the length to be used is the same as the field specified on the REFFLD parameter. For specific information on allowable field lengths see [Field Types](#)

Type	Notes for Length parameter
*REFFLD	*REFFLD, or any value that is valid for the underlying field type of the reference field.
*DEC or synonym *PACKED	LENGTH(*REFFLD) may only be specified if parameter REFFLD is specified. If the REFFLD Parameter is specified, and it is an RDML field, changing the length to 31 or higher will make the working field an RDMLX field.
*CHAR	LENGTH(*REFFLD) may only be specified if parameter REFFLD is specified.

*STRING	If LENGTH(*REFFLD) REFFLD(*NONE) is specified, the length will default to 256.
*SIGNED	LENGTH(*REFFLD) may only be specified if parameter REFFLD is specified. If the REFFLD Parameter is specified, and it is an RDML field, changing the length to 31 or higher will make the working field an RDMLX field.
*BIN	LENGTH(*REFFLD) may only be specified if parameter REFFLD is specified.
*DATE	Dates are fixed size (always 10) incr/decr must be *NONE # to incr/decr must be *NONE
*TIME	Times are fixed size (always 8) incr/decr must be *NONE # to incr/decr must be *NONE
*DATETIME	19, 21-29. The various lengths influence the number of fractional seconds. This must be made clear. A length of 19 means no fractional seconds, 21 - 29 means 1 - 9 fractional seconds. The DECIMALS parameter has no impact. If LENGTH(*REFFLD) REFFLD(*NONE) is specified, the length will default to the ISO standard of 26: YYYY-MM-DD HH:MM:SS.ffffff incr/decr must be *NONE # to incr/decr must be *NONE

"Incr/decr" value is used in conjunction with *REFFLD on the length parameter. The purpose of this field is to allow the length value as obtained from the field specified on the REFFLD keyword to be altered. Permissible values are *PLUS, *MINUS and *NONE. *PLUS specifies that the REFFLD field length attribute is to be increased. *MINUS specifies that the REFFLD field length attribute is

to be decreased. *NONE specifies that the REFFLD field length attribute is to remain the same.

"# to incr/decr" value is used in conjunction with the *REFFLD value on the length parameter and is directly related to the "incr/decr" value. The purpose of this field is to specify the value by which the REFFLD field length value is to be increased or decreased. Permissible values for this field are a numeric value or the value *NONE.

DECIMALS

Specifies the number of decimal positions of the field being defined and is used in conjunction with the type value of *DEC. If the value *REFFLD is specified then the decimal positions to be used are the same as the field specified on the REFFLD parameter. Otherwise a value in the range 0 to 63 must be specified.

All Fields of types other than Signed and Packed must have DECIMALS(0) or DECIMALS(*REFFLD *NONE *NONE) specified.

"Incr/decr" value is used in conjunction with *REFFLD on the decimals parameter. The purpose of this field is to allow the decimal positions value as obtained from the field specified on the REFFLD keyword to be altered. Permissible values are *PLUS, *MINUS and *NONE. *PLUS specifies that the REFFLD field decimal positions attribute is to be increased. *MINUS specifies that the REFFLD field decimal positions attribute is to be decreased. *NONE specifies that the REFFLD field decimal positions attribute is to remain the same.

"# to incr/decr" value is used in conjunction with the *REFFLD value on the decimal positions parameter and is directly related to the "incr/decr" value. The purpose of this field is to specify the value by which the REFFLD field decimal positions value is to be increased or decreased. Permissible values for this field are a numeric value or the value *NONE.

REFFLD

Specifies the name of the field on which this definition is based.

LABEL

Specifies the 15 character label which should be assigned to this field.

*DEFAULT indicates the default label should be used. If the REFFLD parameter is used then the label of the referenced field will be used. If the REFFLD parameter is not used then the name of the field being defined will be used as the label.

DESC

Specifies the 50 character description that should be assigned to this field. *DEFAULT specifies the default description should be used. If the REFFLD parameter is used then the description of the referenced field will be used. If the REFFLD parameter is not used then the name of the field being defined will be used as the description.

COLHDG

Specifies the 3 x 20 character column headings that should be assigned to this field. *DEFAULT specifies that the default column headings should be used. If the REFFLD parameter is specified then the column heading of the referenced field will be used. If REFFLD is not used then the name of the field will be used as column heading 1.

EDIT_CODE

Specifies the edit code (if any) which is to be assigned to the field being defined. If no edit code is specified then the value *DEFAULT is assumed.

*DEFAULT indicates that the edit code of the REFFLD field should be used if the REFFLD parameter is used. Otherwise no edit code should be used for the field.

Use of edit codes for all numeric fields (e.g. type *DEC) is strongly recommended.

Fields of type Integer, Signed, or Packed may have an Editcode or Editword, or may leave both as *DEFAULT. However, Integer does not allow edit codes W and Y. All other field types must have EDIT_CODE(*DEFAULT) EDIT_WORD(*DEFAULT).

Edit codes supported by LANSAs are shown in [Standard Field Edit Codes](#).

EDIT_WORD

Specifies the edit word (if any) which is to be assigned to the field being defined. If no edit word is specified then the value *DEFAULT is assumed.

*DEFAULT indicates that the edit word of the REFFLD field should be used if the REFFLD parameter is used. Otherwise no edit word should be used for the field.

Fields of type Integer, Signed, or Packed may have an Editcode or Editword, or may leave both as *DEFAULT. All other field types must have EDIT_CODE(*SAME) EDIT_WORD(*SAME).

Use of edit words should only be attempted by experienced users as the validity

checking done by LANSAs is unsophisticated.

Note that by using the REFFLD option and EDIT_WORD(*DEFAULT) you are specifying that the edit word associated with the REFFLD should be used.

However, if the length or number of decimal positions used are different to the REFFLD field then the associated edit word may be invalid. In such cases it will be necessary to define the required edit word.

Note also that the operating system handles edit words involving floating currency symbols on screen panels differently to how they are handled on reports. In such cases, it is suggested that a separate field (or a "virtual" field) is used for report production.

When an edit word is defined in LANSAs via the RDML command language it should be enclosed in **triple quotes** as opposed to single quotes.

For example:

Correct Method for defining an edit word for a 5,2 numeric field requiring a trailing %.

```
DEFINE FIELD(#INCREASE) TYPE(#DEC) LENGTH(5) DECIMALS(2) L.
```

Incorrect Method for defining an edit word for a 5,2 numeric field requiring a trailing %.

```
DEFINE FIELD(#INCREASE) TYPE(#DEC) LENGTH(5) DECIMALS(2) L.
```

For further details, refer to keyword EDTWRD in IBM manual *Data Description Specifications*.

INPUT_ATR

Specifies the input attributes which are to be assigned to the field being defined. If no input attributes are defined then the value *DEFAULT is assumed.

*DEFAULT indicates that the input attributes of the REFFLD field should be used if the REFFLD parameter is used. Otherwise the system default input attributes list for either alpha or numeric fields should be used according to the field type.

For information on allowable attributes for RDMLX fields see [Field Types](#)

Valid input attributes for types A (alphanumeric), P (packed), and S (signed) are:

Attribute	Description / Comments	A	P	S
AB	Allow to be blank.	Y	Y	Y

ME	Mandatory entry check required.	Y	Y	Y
MF	Mandatory fill check required.	Y	Y	Y
M10	Modulus 10 check required.		Y	Y
M11	Modulus 11 check required.		Y	Y
VN	Valid name check required.	Y		
FE	Field exit key required.	Y	Y	Y
LC	Lowercase entry allowed. If you do NOT set this attribute, refer to <i>PC Locale uppercasing requested</i> in Review or Change a Partition's Multilingual Attributes in the <i>LANSA for i User Guide</i> .	Y		
RB	Right adjust and blank fill.		Y	Y
RZ	Right adjust and zero fill.		Y	Y
RL	Move cursor right to left.	Y	Y	Y
RLTB	Tab cursor right/left top/bottom. Valid in SAA/CUA partitions only. Affects all screen panels	Y	Y	Y
GRN	Display with color green.	Y	Y	Y
WHT	Display with color white.	Y	Y	Y
RED	Display with color red.	Y	Y	Y
TRQ	Display with color turquoise.	Y	Y	Y
YLW	Display with color yellow.	Y	Y	Y
PNK	Display with color pink.	Y	Y	Y
BLU	Display with color blue.	Y	Y	Y
BL	Display blinking.	Y	Y	Y
CS	Display with column separators.	Y	Y	Y
HI	Display in high intensity.	Y	Y	Y
ND	Non-display (hidden field).	Y	Y	Y
RA	Auto record advance field	Y	Y	Y

SREV	Store in reversed format. This special attribute is provided for bi-directional languages & is not applicable in this context.	Y	N	N
SBIN	Store in binary format. This special attribute is provided for repository fields & is not applicable in this context.	Y	N	N
HIND	HINDI Numerics. Display using HINDI numerals. Refer to Hindi Numerics in the <i>LANSA for i User Guide</i> .	N	Y	Y
CBOX *	Check Box	Y	N	N
RBnn *	Radio Button	Y	N	N
PBnn *	Push Button	Y	N	N
DDXX *	Drop Down.	Y	N	N

Attributes marked with * represent the field with the corresponding GUI WIMP construct. Refer to [GUI WIMP Constructs](#) in the *LANSA for i User Guide* for more information.

In partitions that comply with **SAA/CUA guidelines** the following attributes may be used as well (and are in fact preferred to those described above):

Attribute	Description / Comments
ABCH	Action bar and pull-down choices
PBPT	Panel title
PBPI	Panel identifier
PBIN	Instructions to user
PBFP	Field prompt / label / description details
PBBR	Brackets
PBCM	Field column headings
PBGH	Group headings
PBNT	Normal text

PBET	Emphasized text
PBEN *	Input capable field (normal)
PBEE *	Input capable field (emphasized)
PBCH	Choices shown on menu
PBSC	Choice last selected from menu
PBUC	Choices that are not available
PBCN	Protected field (normal)
PBCE	Protected field (emphasized)
PBSI	Scrolling information
PBSL	Separator line
PBWB	Pop-up window border
FKCH	Function key information

Note: Normally only PBEN and PBEE would be specified as input attributes. Refer to *SAA/CUA Implementation* in the *LANSA Application Design Guide* for more details of these attributes. Also note that only one color can be specified for a field. Use of colors may affect other attributes. Refer to IBM manual *Data Description Specifications* for more details. Keywords that should be reviewed are CHECK, COLOR and DSPATR.

OUTPUT_ATR

Specifies the output attributes which are to be assigned to the field being defined. If no output attributes are specified then the value *DEFAULT is assumed.

*DEFAULT indicates that the output attributes of the REFFLD field should be used if the REFFLD parameter is used. Otherwise the system default output attributes list for either alpha or numeric fields should be used according to the field type.

For information on allowable attributes for RDMLX fields see [Field Types](#)
Valid output attributes for types Alpha (A), Packed (P), and Signed (S) are:

Attribute	Description / Comments	A	P	S
GRN	Display with color green.	Y	Y	Y
WHT	Display with color white.	Y	Y	Y
RED	Display with color red.	Y	Y	Y
TRQ	Display with color turquoise.	Y	Y	Y
YLW	Display with color yellow.	Y	Y	Y
PNK	Display with color pink.	Y	Y	Y
BLU	Display with color blue.	Y	Y	Y
BL	Display blinking.	Y	Y	Y
CS	Display with column separators.	Y	Y	Y
HI	Display in high intensity.	Y	Y	Y
ND	Non-display (hidden field).	Y	Y	Y
SREV	Store in reversed format. This special attribute is provided for bi-directional languages and is not applicable in this context.	Y	N	N
SBIN	Store in binary format. This special attribute is provided for repository fields & is not applicable in this context.	Y	N	N
Urxx	User Defined Reporting Attribute. Provides access to IBM i DDS statements for printer files. Refer to User Defined Reporting Attributes in the <i>LANSA for i User Guide</i> .	Y	Y	Y
HIND	HINDI Numerics. Display using HINDI numerals. Refer to Hindi Numerics in the <i>LANSA for i User Guide</i> .	N	Y	Y
CBOX *	Check Box	Y	N	N
RBnn *	Radio Button	Y	N	N
PBnn *	Push Button	Y	N	N
DDxx *	Drop Down.	Y	N	N

Attributes marked with an * represent the field with the corresponding GUI WIMP construct. Refer to [GUI WIMP Constructs](#) in the *LANSAs for i User Guide* for more information

In partitions that comply with **SAA/CUA guidelines**, the following attributes may be used as well (and are in fact preferred to those described above):

Attribute	Description / Comments
ABCH	Action bar and pull-down choices
PBPT	Panel title
PBPI	Panel identifier
PBIN	Instructions to user
PBFP	Field prompt / label / description details
PBBR	Brackets
PBCM	Field column headings
PBGH	Group headings
PBNT	Normal text
PBET	Emphasized text
PBEN	Input capable field (normal)
PBEE	Input capable field (emphasized)
PBCH	Choices shown on menu
PBSC	Choice last selected from menu
PBUC	Choices that are not available
PBCN *	Protected field (normal)
PBCE *	Protected field (emphasized)
PBSI	Scrolling information
PBSL	Separator line
PBWB	Pop-up window border
FKCH	Function key information

*** Note:** Normally only PBCN and PBCE would be specified as output attributes. Refer to *SAA/CUA Implementation* in the *LANSA Application Design Guide* for more details of these attributes. Also note that only one color can be specified for a field. Use of colors may affect other attributes. Refer to IBM manual *Data Description Specifications* for more details. Keywords that should be reviewed are COLOR and DSPATR.

DEFAULT

Specifies the default value which is to apply to the field being defined.

This is the value that the field will contain when the function begins to execute. Note that using the EXCHANGE command can appear to alter the default value of a field.

For information on what DEFAULT(*DEFAULT) means for RDMLX fields see [Field Types](#)

If no default value is specified then *DEFAULT is assumed. This means that if the REFFLD parameter has been specified the default value of the REFFLD field will be used. If the REFFLD parameter has not been used then a default value of *BLANKS will be used for alphanumeric fields and a default value of *ZERO for numeric fields will be used.

Default values specified can be:

- A system variable such as *BLANKS, *ZERO, *DATE or any other specifically defined at your installation.
- An alphanumeric literal such as BALMAIN.
- A numeric literal such as 1, 10.43, -1.341217.
- A process parameter such as *UP01.
- *SQLNULL is allowed as a default value for any field type that has ASQN as an input or output attribute.
- *NULL is allowed as a default value for any field type (if the partition is RDMLX enabled).

TO_OVERLAY

Specifies that the field being defined is to fully or partially overlay (i.e. occupy the same storage locations) as the field referenced in this parameter.

It is invalid for RDMLX fields to be overlaid or overlay another field.

*NONE, which is the default value, indicates that the field being defined is to occupy its own storage area and not to overlay any other field.

The only other allowable value that can be specified here is the name of another field defined in this program or the data dictionary, optionally followed by a starting position.

The TO_OVERLAY parameter is a powerful facility that allows a field to occupy the same storage (ie: memory locations) as another field. The power of this parameter means that you must understand exactly what it causes to happen and what problems you may cause yourself in using it.

The following notes and comments should be read in full before attempting to use this parameter:

- You must **NOT** overlay a field onto a field that is itself overlaid onto another field. This is NOT checked by the full function checker and may cause a compile failure.
- You should fully understand the IBM i data storage formats of **character**, **signed/zoned decimal** and **packed decimal** before attempting to overlay fields of varying types. Overlaying of fields means that you can easily cause invalid decimal data to be placed into decimal fields, thus causing your program to fail in an unpredictable manner.
- Array index fields must not be overlaid on or by other fields (in any context).
- The start position component of this parameter allows you to overlay just a part of a specific field, rather than its entire length. The start position is a full byte position, even when using packed decimal fields. When you specify a start position you **MUST** ensure that you do not overlay the field beyond the end position of the field being overlaid.

This is **NOT** checked by the full function checker. Failure to observe this rule can cause dangerous and unpredictable results.

- A packed decimal field of even length can be overlaid on another field, however the RPG compiler will always interpret the overlaying field as the next highest odd length. For example:

```
DEFINE FIELD(#DEC6) TYPE(*DEC) LENGTH(6) DECIMALS(0)
DEFINE FIELD(#OVR6) TYPE(*DEC) LENGTH(6) DECIMALS(0) TO_OV
```

will cause #DEC6 to be treated by the RPG compiler as a packed decimal (6,0) value. However, #OVR6 will be treated by the RPG compiler as a

packed decimal (7,0) value. There is no memory length problem here, both fields require 4 bytes of memory to be stored, it is just the way that the RPG compiler works that may cause a presentation length problem on reports. HOWEVER, if #OVR6 is put on a screen as only *OUTPUT, the function compile will crash. This is because the external description of #OVR6 from the display file will say that it is 6 digits, packed. Meanwhile, as stated above, the overlay causes the RPG compiler to assume that #OVR6 is 7 digits, packed.

- When the data validation commands RANGECHECK, VALUECHECK, DATECHECK, CALLCHECK, CONDCHECK, FILECHECK or SET_ERROR are used on an overlaying field, they also set an error for the overlaid field.

For example:

```
DEFINE FIELD(#INPUT) TYPE(*CHAR) LENGTH(3)
DEFINE FIELD(#INPC1) TYPE(*CHAR) LENGTH(1)    TO_OVERLA
DEFINE FIELD(#INPC3) TYPE(*CHAR) LENGTH(1)    TO_OVERLA

REQUEST FIELDS(#INPUT)

BEGINCHECK
VALUECHECK FIELD(#INPC1) WITH_LIST('A' 'B' 'C')
VALUECHECK FIELD(#INPC3) WITH_LIST('X' 'Y' 'Z')
ENDCHECK
```

This program accepts a 3 character field (#INPUT) from the workstation and validates that the first character is an A, B or C and also that the last character is an X, Y or Z.

When an error is triggered against overlaid fields #INPC1 or #INPC3 by the VALUECHECK commands, it is also triggered against the overlaid field #INPUT. This means that when the REQUEST command is (re)executed in an error situation, field #INPUT will be displayed in reverse video.

SHIFT

Specifies the keyboard shift (if any) which is to be assigned to the field being defined. If no keyboard shift is specified then the value *DEFAULT is assumed.

*DEFAULT indicates that the keyboard shift of the REFFLD field should be used if the REFFLD parameter is used. Otherwise no keyboard shift should be used for the field.

For information on what values of SHIFT, apart from *DEFAULT, are valid for each working field type see [Field Types](#).

For working fields of type Boolean, SHIFT must be *DEFAULT.

Refer to the IBM manual *Data Description Specifications* for more details. Position 35 for display files is the entry that should be reviewed.

7.25.2 DEFINE Examples

Define a work / counter field #I for internal use in an RDML program

Define a work / counter field #I for internal use

Define a field called #LASTORDER with same #ORDER attributes

Define a field called #LASTORDER with different #ORDER attributes

Define a field called #TOTQTY with 3 more significant digits than #QTY

Define a field called #TOTQTY with different attributes to #QTY

Define a field called #SHORT with similar attributes to field #LONG

Define a field called #SHORT with similar attributes to #LONG

Define a work / counter field #I for internal use in an RDML program

```
DEFINE FIELD(#I) TYPE(*DEC) LENGTH(7) DECIMALS(0)
```

Define a work / counter field #I for internal use

Define in an RDML program so that it will contain value 2 when the function begins to execute:

```
DEFINE FIELD(#I) TYPE(*DEC) LENGTH(7) DECIMALS(0) DEFAULT(2)
```

Define a field called #LASTORDER with same #ORDER attributes

Ensure that it has exactly the same attributes as field #ORDER which is defined in the LANSAs data dictionary.

```
DEFINE FIELD(#LASTORDER) REFFLD(#ORDER)
```

Define a field called #LASTORDER with different #ORDER attributes

Ensure that it has exactly the same attributes as field #ORDER except for the description, label and column headings.

```
DEFINE FIELD(#LASTORDER) REFFLD(#ORDER) DESC('Last Order Nur
```

Define a field called #TOTQTY with 3 more significant digits than #QTY

Ensure that it has exactly the same attributes as field #QTY except for having 3 more significant digits.

```
DEFINE FIELD(#TOTQTY) REFFLD(#QTY) LENGTH(*REFFLD *PLUS 3
```

Define a field called #TOTQTY with different attributes to #QTY

Ensure it has exactly the same attributes as field #QTY except for having 3 more significant digits and 2 more decimal digits.

```
DEFINE FIELD(#TOTQTY) REFFLD(#QTY) LENGTH(*REFFLD *PLUS 5
```

Define a field called #SHORT with similar attributes to field #LONG

Make sure #SHORT is exactly 10 characters long.

```
DEFINE FIELD(#SHORT) REFFLD(#LONG) LENGTH(10)
```

Define a field called #SHORT with similar attributes to #LONG

Make sure #SHORT has exactly the same attributes as field #LONG except that it is 10 characters shorter.

```
DEFINE FIELD(#SHORT) REFFLD(#LONG) LENGTH(*REFFLD *MINUS
```

Define a numeric field based on another field

Define it with the S keyboard shift so that it can be displayed with edit code J.

```
DEFINE FIELD(#SHIFTY) REFFLD(#SHIFTS) EDIT_CODE(J) SHIFT(Y)
```

7.26 DELETE

The DELETE command allows the user to delete records from a file either by key or relative record number.

Portability Considerations Refer to parameter [FROM_FILE](#).

Also See

[7.26.1 DELETE Parameters](#)

[7.26.2 DELETE Comments / Warnings](#)

[7.26.3 DELETE Examples](#)

Required

```
DELETE ----- FROM_FILE ---  
- file name . library name ----->
```

Optional

```
>-- WHERE ----- 'condition' ----->  
  
>-- WITH_KEY ----- key value ----->  
    | expandable group expression |  
    --- 20 maximum -----  
  
>-- IO_STATUS ----- *STATUS ----->  
    field name  
  
>-- IO_ERROR ----- *ABORT -----  
>  
    *NEXT  
    *RETURN  
    label  
  
>-- VAL_ERROR ----- *LASTDIS -----  
>  
    *NEXT  
    *RETURN
```

label

```
>-- NOT_FOUND ---- *NEXT -----  
->  
    *RETURN  
    label  
  
>-- ISSUE_MSG ---- *NO ----->  
    *YES  
  
>-- WITH_RRN ---- *NONE -----  
>  
  
>-- RETURN_RRN --- *NONE -----  
->  
  
>-- CHECK_ONLY --- *NO -----  
>  
    *YES  
  
>-- AUTOCOMMIT --- *FILEDEF -----  
---|  
    *YES  
    *NO
```

7.26.1 DELETE Parameters

AUTOCOMMIT

CHECK_ONLY

FROM_FILE

IO_ERROR

IO_STATUS

ISSUE_MSG

NOT_FOUND

RETURN_RRN

WHERE

VAL_ERROR

WITH_KEY

WITH_RRN

FROM_FILE

Refer to [Specifying File Names in I/O Commands](#).

WHERE

Refer to [Specifying Conditions and Expressions](#) and [Specifying WHERE Parameter in I/O Commands](#).

WITH_KEY

Refer to [Specifying File Key Lists in I/O Commands](#).

For details of how using this parameter can affect automatic "cross update" checking, refer to the Delete Comments/Warnings section.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

For values, refer to [I/O Return Codes](#).

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command. The purpose of *NEXT is to permit you to handle error messages in the RDML, and then ABORT, rather than use the default ABORT. (It is possible for processing to continue for LANSAs for i and Visual LANSAs, but this is NOT a recommended way to use LANSAs.)

ER returned from a database operation is a fatal error and LANSAs does not expect processing to continue. The IO Module is reset and further IO will be as if no previous IO on that file had occurred. Thus you must not make any presumptions as to the state of the file. For example, the last record read will not be set. A special case of an IO_ERROR is when a trigger function is coded to return ER in TRIG_RETC. The above description applies to this case as well. Therefore, LANSAs recommends that you do NOT use a return code of ER from a trigger function to cause anything but an ABORT or EXIT to occur before any further IO is performed.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

VAL_ERROR

Specifies the action to be taken if a validation error was detected by the command.

A validation error occurs when information that is to be added, updated or deleted from the file does not pass the FILE or DICTIONARY level validation checks associated with fields in the file.

If the default value *LASTDIS is used control will be passed back to the last display screen used. The field(s) that failed the associated validation checks will

be displayed in reverse image and the cursor positioned to the first field in error on the screen.

*NEXT indicates that control should be passed to the next command.

*RETURN indicates that control should be returned to the invoking routine (identical to executing a RETURN command).

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).

When using *LASTDIS the "Last Display" must be at the same level as the database command (INSERT, UPDATE, DELETE, FETCH and SELECT). If they are at different levels e.g. the database command is specified in a SUBROUTINE, but the "Last Display" is a caller routine or the mainline, the function will abort with the appropriate error message(s).

The same does NOT apply to the use of event routines and method routines in Visual LANSA. In these cases, control will be returned to the calling routine. The fields will display in error with messages returned to the first status bar encountered in the parent chain of forms, or if none exist, the first form with a status bar encountered in the execution stack (for example, a reusable part that inherits from PRIM_OBJT).

NOT_FOUND

Specifies what is to happen if no record is found in the file that has a key matching the key nominated in the WITH_KEY parameter.

*NEXT indicates that control should be passed to the next command.

*RETURN indicates that control should be returned to the invoking routine (identical to executing a RETURN command).

If none of the previous values are used you must nominate a valid command label to which control should be passed.

ISSUE_MSG

Specifies whether a "not found" message is to be automatically issued or not.

The default value is *NO which indicates that no message should be issued. The only other allowable value is *YES which indicates that a message should be automatically issued. The message will appear on line 22/24 of the next screen format presented to the user or on the job log of a batch job.

WITH_RRN

Specifies the name of a field that contains the relative record number (for relative record file processing) of the record which is to be deleted. The WITH_RRN parameter cannot be used if the WITH_KEY or WHERE parameters are used.

Any field nominated in this parameter must be defined within the function or the LANSAs data dictionary and must be numeric.

Note: Using the WITH_RRN parameter to FETCH, DELETE or UPDATE records is faster than any other form of database access.

The actual database file being accessed is always the physical file, regardless of whether or not the file nominated in the command is a logical file. Thus logical file select/omit criteria are not used when accessing a logical file via the WITH_RRN parameter.

Refer also to:

- [7.26.2 DELETE Comments / Warnings](#) for details of how using this parameter can affect automatic "cross update" checking.
- [Load Other File](#) in the *Visual LANSAs Developers Guide*.

RETURN_RRN

Specifies the name of a field in which the relative record number of the record just deleted should be returned. The value returned in this field is not usable when the DELETE command is deleting multiple records from the file.

Any field nominated in this parameter must be defined within the function or the LANSAs data dictionary and must be numeric.

For further information refer also to [Load Other File](#) in the *Visual LANSAs Developers Guide*.

CHECK_ONLY

Indicates whether the I/O operation should actually be performed or only "simulated" to check whether all file and data dictionary level validation checks can be satisfied when it is actually performed.

*NO, which is the default value, indicates that the I/O operation should be performed in the normal manner.

*YES indicates that the I/O operation should be simulated to verify that all file and data dictionary level checks can be satisfied. The **database** file involved is not changed in any way when this option is used.

AUTOCOMMIT

This parameter was made redundant in LANSAs release 4.0 at program change level E5.

To use commitment control specify COMMIT and/or ROLLBACK commands in your application.

Generally only COMMIT commands are required.

For the implications of using commitment control on the IBM i, refer to [Commitment Control](#) in the *LANSAs for i User Guide*.

Portability If using Visual LANSAs, refer to [Commitment Control](#) in the
Considerations *LANSAs Application Design Guide*.

7.26.2 DELETE Comments / Warnings

- The use of automatic "**crossed update**" checks by the DELETE command should be clearly understood.

Consider the following flow of commands:

```
FETCH WITH_KEY() or WITH_RRN()  
DISPLAY  
IF_MODE *DELETE  
DELETE  
ENDIF
```

Since the DELETE command has **no** WITH_KEY or WITH_RRN parameter it is indicating that the **last record read** (by the FETCH command) should be deleted.

In this situation, the "**crossed update window**" is in the interval between the time the record was FETCHed and the time that it is DELETED. This could be **very long** if the user went and had a cup of coffee when the DISPLAY command was on his/her workstation.

This is a **correct and valid** use of the automatic "crossed update" checking facility. If the record was changed by another job/user between the FETCH and the DELETE, then the DELETE will generate a "crossed update error" (which should be handled just like any other type of validation error).

Now consider this flow of commands:

```
FETCH WITH_KEY() or WITH_RRN()  
DISPLAY  
IF_MODE *DELETE  
DELETE WITH_KEY() or WITH_RRN()  
ENDIF
```

Since the DELETE command **has** a WITH_KEY or WITH_RRN parameter it is indicating that a specific record (or group of records) should be **read** and **deleted**.

This is a common coding mistake. Everybody knows that the WITH_KEY or WITH_RRN values on the DELETE command should/would be the same as those on the FETCH command. However, the RDML compiler cannot be sure that the values were not changed, so it is **forced** to (re)read the record before

attempting the DELETE.

In this situation, the "**crossed update window**" is in the interval between the time the record is (re)read by the DELETE command and then deleted by the DELETE command. This interval is very short, and thus the "crossed update" check is effectively disabled.

This is **not** considered to be a **valid and correct** use of the DELETE command in an **interactive program** like this because it effectively disables the automatic "crossed update" check.

- Where a DELETE operation is issued with no WITH_KEY, WHERE or WITH_RRN parameters specified the last record read from the file will be deleted. Thus the following are equivalent operations:

```
DELETE FROM_FILE(ORDHDR) WITH_KEY(#ORDNUM)
```

is functionally equivalent to:

```
FETCH FROM_FILE(ORDHDR) WITH_KEY(#ORDNUM)  
DELETE FROM_FILE(ORDHDR)
```

and:

```
DELETE FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM)
```

is functionally equivalent to:

```
SELECT FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM)  
DELETE FROM_FILE(ORDLIN)  
ENDSELECT
```

7.26.3 DELETE Examples

Example 1: Delete an order specified in field #ORDNUM from an order header file:

```
DELETE FROM_FILE(ORDHDR) WITH_KEY(#ORDNUM)
```

Example 2: Delete order line number 1 and then order line 2 from an order lines file called ORDLIN. The order number is contained in a field called #ORDNUM:

```
DELETE FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM 1)
DELETE FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM 2)
```

Note the use of the numeric literals 1 and 2 as key values.

This example could also have been coded as:

```
CHANGE FIELD(#I) TO(1)
DOWHILE COND('#I <= 2')
DELETE FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM #I)
CHANGE FIELD(#I) TO('#I + 1')
ENDWHILE
```

Example 3: Delete all order lines associated with an order specified in field #ORDNUM:

```
DELETE FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM)
```

Note that this command deletes multiple records from the file.

Example 4: Delete all order lines associated with an order specified in field #ORDNUM and then delete the order header record:

```
DELETE FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM)
DELETE FROM_FILE(ORDHDR) WITH_KEY(#ORDNUM)
```

Example 5: Delete all records from file NAMES where field #DLTIND contains a Y:

```
DELETE FROM_FILE(NAMES) WHERE('#DLTIND = Y')
```

Example 6: Delete all records from file NAMES where field #DLTIND

contains a Y and field #UPDATE is less than the current date:

```
DELETE FROM_FILE(NAMES) WHERE('#DLTIND = Y) AND (#UPDATI
```

Note the use of the system variable *YYMMDD which contains the current date in format YYMMDD (which is presumably the same format as #UPDATE).

7.27 DISPLAY

The DISPLAY command allows the user to display information on a workstation.

The DISPLAY command is only valid in RDMLX functions when being used on the Web. If it is used elsewhere a fatal error occurs at runtime. If this occurs, either put your DISPLAY command in an RDML function or use a Form to show user information.

Portability Considerations Refer to parameters: [FIELDS](#), [IGCCNV_KEY](#), [OPTIONS](#), [SHOW_NEXT](#), and [TEXT](#)

Also See

[7.27.1 DISPLAY Parameters](#)

[7.27.2 DISPLAY Comments / Warnings](#)

[7.27.3 DISPLAY Examples](#)

Optional

DISPLAY ----- FIELDS ----- field name field attributes -
-->

```
|          |          ||
|          --- 7 max ----- |
| expandable group expression |
|----- 1000 max for RDMLX----|
|----- 100 max for RDML -----
```

```
>-- DESIGN ----- *IDENTIFY ----->
    *DOWN
    *ACROSS
```

```
>-- IDENTIFY ----- *DESIGN ----->
    *COLHDG
    *LABEL
    *DESC
    *NOID
```

```
>-- IDENT_ATR ----- *DEFAULT ----->
-->
```

**NONE*
**HI *RI *UL (3 maximum)*

*>-- DOWN_SEP ----- *DESIGN -----*
->
decimal value

*>-- ACROSS_SEP --- *DESIGN -----*
->
decimal value

*>-- BROWSELIST --- *NONE ----- 999 -----*
->
name of list no.entries/page

*>-- EXIT_KEY ----- *YES -- *EXIT -*
*- *HIGH - *NONE ->*
**NO *MENU *LOW condition*
**NEXT*
**RETURN*
label

*>-- MENU_KEY ----- *YES -- *MENU ---*
*- *NONE ----->*
**NO *EXIT condition*
**RETURN*
**NEXT*
label

*>-- ADD_KEY ----- *NO --- *NEXT ---- *NONE ---*
---->
**YES *RETURN condition*
label

*>-- CHANGE_KEY --- *NO --- *NEXT ---*
*- *NONE ----->*
**YES *RETURN condition*
label

>-- DELETE_KEY --- *NO --- *NEXT ---- *NONE -
----->

*YES *RETURN condition
label

>-- PROMPT_KEY --- *DFT -- *AUTO ---
- *NONE ----->

*YES *NEXT condition
*NO *RETURN
label

>-- USER_KEYS --- fnc key--'desc'--*NEXT -
- cond -->

```
|          *RETURN      |  
|          label        |  
|                          |  
----- 5 maximum -----
```

>-- PANEL_ID ----- *AUTO ----->
or *NONE
or panel identifier

>-- PANEL_TITL --- *FUNCTION -----
-->
or 'Panel title'

>-- SHOW_NEXT ---- *PRO -----
>
*YES
*NO

>-- TEXT ----- 'text' --- line/ --- position -->
| row column |
----- 50 max -----
*TMAPnnn 1 1 (special value)

>-- CURSOR_LOC --- *NONE ----- *NONE -----
----->
*ATFIELD field name

row value column value

>-- *STD_HEAD* ----- **DFT* ----->

**YES*

**NO*

>-- *OPTIONS* ----- **NONE* ----->

**NOREAD *OVERLAY (2 maximum)*

>-- *IGCCNV_KEY* -- **AUTO* -----

**YES*

**NO*

condition name

7.27.1 DISPLAY Parameters

ACROSS_SEP
ADD_KEY
BROWSELIST
CHANGE_KEY
CURSOR_LOC
DELETE_KEY
DESIGN
DOWN_SEP
EXIT_KEY
FIELDS
IDENT_ATR
IDENTIFY
IGCCNV_KEY
MENU_KEY
OPTIONS
PANEL_ID
PANEL_TITL
PROMPT_KEY
SHOW_NEXT
STD_HEAD
TEXT
USER_KEYS

FIELDS

Specifies either the field(s) that are to be displayed at the workstation or the name of a group that specifies the field(s) to be displayed. Alternatively, an expandable group expression can be entered in this parameter.

Portability Visual LANSA has multi-page and field spanning line
Considerations restrictions:

Multi-page data (i.e. if the screen format is larger than one page) can be displayed in a Web browser window but NOT in

a LANSAs function.

If a process containing multi-page data is compiled, a warning will be issued if the process is WEB/XML enabled. If the process is NOT WEB/XML enabled, a full function check error will be issued.

Field spanning (i.e. when the field is larger than one line on the screen) is not supported - only a single line will be displayed. No error or warning is issued.

DESIGN

Specifies the design/positioning method which should be used for fields that **do not have specific positioning attributes** associated with them.

*IDENTIFY, which is the default value, indicates that the design method should be the default method associated with the IDENTIFY parameter. Refer to the table in the Comments section for more details.

*DOWN indicates that the fields should be designed "down" the screen in a column.

*ACROSS indicates that fields should be designed "across" the screen in a row.

IDENTIFY

Specifies the default identification method to be used for fields that **do not have a specific identification attribute** associated with them.

*DESIGN, which is the default value, indicates that the fields should be identified by the default method associated with the DESIGN parameter. See the table in the comments section for more details.

*LABEL indicates that fields should be identified by their associated labels on the screen.

*DESC indicates that fields should be identified by their associated descriptions on the screen.

*COLHDG indicates that fields should be identified by their associated column headings on the screen.

*NOID indicates that no identification of the field is required. Only the field itself should be included into the screen design.

IDENT_ATR

Specifies display attributes that are to be associated with identification text (labels, descriptions, column headings, etc) that are displayed on the screen.

*DEFAULT, which is the default value, indicates that the system defaults for identification display attributes should be adopted. They are set up in the system definition block as overall system default values. Refer to [The System Definition Data Areas](#) in the *LANSAs for i User Guide* and how to change it.

*NONE indicates that identification text should have no special display attributes associated with it.

Otherwise, specify one or more of the values: *HI (high intensity), *RI (reverse image) and *UL (underline).

This parameter is **ignored in SAA/CUA processes in SAA/CUA compliant partitions**. In such partitions the attributes are determined from the partition wide standards for labels and column headings.

DOWN_SEP

Specifies the spacing between rows on the display that should be used when automatically designing a screen. The value specified must be *DESIGN or a number in the range 1 to 10. Refer to the table in the Comments section for details of what value *DESIGN is actually specifying.

ACROSS_SEP

Specifies the spacing between columns on the display that should be used when automatically designing a screen. The value specified must be *DESIGN or a number in the range 1 to 10. Refer to the table in the Comments section for details of what value *DESIGN is actually specifying.

BROWSELIST

Specifies the name of a browse list which is also to be included into the screen format, and optionally, the number of entries of the browse list that should appear in the screen panel.

*NONE indicates that no browse list is required. The screen designed will not have any browse component.

If a browse list is specified, then you may also specify the number of entries from the browse list that are to appear on the screen panel. This may leave space below the browse list for other details (which can be overlaid by a subsequent screen). The default of 999 entries indicates that the browse list should extend to the logical bottom of the screen panel.

If a browse list is specified it must be defined elsewhere in the RDML program with a DEF_LIST (define list) command.

EXIT_KEY

Specifies the following things about the EXIT function key:

- Whether the EXIT function key is to be enabled.
- What is to happen when the EXIT function key is used.
- In SAA/CUA partitions, which EXIT function key is required.
- A condition to control when the EXIT function key is enabled.

By default the EXIT function key is enabled. To disable the EXIT function key specify *NO as the first value for this parameter.

If the EXIT function key is enabled, you may specify what happens when it is used. The allowable values for this second component of the EXIT_KEY parameter are as follows:

- *EXIT The application should exit completely from LANSAs (identical to executing an EXIT command).
- *MENU The process's main menu should be re-displayed (identical to executing a MENU command).
- *NEXT Indicates that control should be passed to the next command.
- *RETURN Specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The value *EXIT is the default for this parameter value.

The default value is *HIGH for this parameter value.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

Note: In SAA/CUA applications it is recommended that only the following 2 variations of the EXIT_KEY parameter are used:

EXIT_KEY(*YES *EXIT *HIGH) in a "main program"

*

EXIT_KEY(*YES *RETURN *LOW) in "subroutines"

MENU_KEY

Specifies whether the MENU function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the MENU key is used.

*NO indicates that the MENU function key should not be enabled when the screen is displayed.

*YES, which is the default value, indicates that the MENU key should be enabled when the screen is displayed. If *YES is used it is also possible to specify the action to be taken when the menu key is used.

*MENU, the default value, specifies that the process's main menu should be re-displayed. *EXIT specifies that the application should exit completely from LANSAs.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

ADD_KEY

Specifies whether the ADD function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the ADD key is used.

*NO, which is the default value, indicates that the ADD function key should not be enabled when the screen is displayed.

*YES indicates that the ADD key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to

which control should be passed when the ADD key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

CHANGE_KEY

Specifies whether the CHANGE function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the CHANGE key is used.

*NO, which is the default value, indicates that the CHANGE function key should not be enabled when the screen is displayed.

*YES indicates that the CHANGE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the CHANGE key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

DELETE_KEY

Specifies whether the DELETE function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the DELETE key is used.

*NO, which is the default value, indicates that the DELETE function key should not be enabled when the screen is displayed.

*YES indicates that the DELETE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the DELETE key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

PROMPT_KEY

Specifies whether the PROMPT function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the PROMPT key is used.

*DFT, which is the default value, indicates that the PROMPT function key should be enabled or disabled automatically according to its default value defined in the system definition data area DC@A01. Refer [System Definition Data Area DC@A01](#) in the *LANSAP for i User Guide* for more information about this default value.

*YES indicates that the PROMPT key should be enabled when the screen is displayed.

*NO indicates that the PROMPT key should NOT be enabled when the screen is displayed.

In any case, when the PROMPT function key is enabled (either by specifying *DFT or *YES for the first part of this parameter), it is possible to also specify what is to happen if the function key is used. Allowable values for this part of

the parameter are:

*AUTO indicates that the prompt key processing should be handled automatically by LANSAs. Before attempting to use this option, refer to [Prompt Key Processing](#).

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

USER_KEYS

Specifies up to 5 additional user function keys that can be enabled when the screen format is displayed at the workstation.

Any user function keys assigned must not conflict with function keys assigned to the standard LANSAs functions of EXIT, MENU, MESSAGES, PROMPT, ADD, CHANGE or DELETE when they are enabled on a command (ie: a function key cannot be assigned to more than one function).

Additional user function keys are specified in the format:

```
(fnc key number) 'description' *NEXT *NONE
*RETURN cond name
Label
```

where -

Fnc key number: Is the function key number in the range 1 to 24 or one of the special values *ROLLUP (roll up key) or *ROLLDOWN (roll down key).

'description' Is a description of the function assigned to the function key. This

description will be displayed on line 23 of the screen format.
Maximum length is 8 characters.

- *NEXT Is the default and indicates that the next command (after this one) should receive control.
- *RETURN Indicates that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.
- Label Indicates the command label to which control should be passed if the command key is used.
- *NONE Indicates that no condition applies to control when the function key is to be enabled or disabled.
- Cond name Indicates that a condition defined by a DEF_COND command should be evaluated to determine whether to enable or disable the function key.

Refer to the IF_KEY command for details of how the function key that was used can be tested in the RDML program.

As an example of use consider the following:

```
DISPLAY FIELDS(#PRODUCT) USER_KEYS((14 'Commit')(15 'Purge'))
  IF_KEY WAS(*USERKEY1)
*   << Commit logic >>
  ENDIF
  IF_KEY WAS(*USERKEY2)
*   << Purge logic >>
  ENDIF
```

Note: The IF_KEY command refers to the keys by symbolic names that indicate the order they are declared in the USER_KEYS parameter, not the actual function key numbers assigned to them. This makes changing function key assignments easier.

PANEL_ID

Specifies the identifier that is to be assigned to the panel or pop-up window created by this command.

*AUTO indicates that it should be automatically generated by LANSAs from the function name and the source statement number of the RDML program.

*NONE indicates that no panel identifier is required for this panel or pop-up window.

Otherwise specify a panel identifier from 1 to 10 characters in length. The value specified is fixed and cannot be changed at execution time.

This parameter is valid for **SAA/CUA applications only**.

This parameter is **ignored** if the current partition definition indicates that panel identifiers are never required, no matter what value is specified.

PANEL_TITL

Specifies the title that is to be assigned to the window panel.

*FUNCTION indicates that it should be derived from the RDML function's description.

Otherwise specify a panel title from 1 to 40 characters in length. The value specified is fixed and cannot be changed at execution time.

This parameter is valid for **SAA/CUA applications only**.

SHOW_NEXT

Specifies whether the "next function" field should be shown on line 22 of the screen. The next function field is a facility that allows transfer between the functions in a process without the need to return to the process menu each time. Refer to the section that describes the [The Function Control Table](#) in the *LANSA for i User Guide* for more details about "next function" processing.

*PRO, which is the default value, indicates that the "next function" field should appear only when the process to which this function belongs has a menu selection style of "FUNCTION". If the process menu selection style is "NUMBER" or "CURSOR" then the next function field should not appear.

*YES indicates that the next function field should appear regardless of what menu selection style is being used by the process to which this function belongs.

*NO indicates that the next function field should not appear regardless of what menu selection style is being used by the process to which this function belongs.

Note: The SHOW_NEXT parameter is ignored in SAA/CUA applications.

Portability Considerations

Ignored in Visual LANSA applications with no known effect to the application.

TEXT

Allows the specification of up to 50 "**text strings**" that are to appear on the

screen panel or report. Each text string specified is restricted to a maximum length of 20 characters.

When a text string is specified it should be followed by a row/line number and a column/position number that indicates where it should appear on the screen panel or report.

For example:

```
TEXT(('ACME' 6 2)('ENGINEERING' 7 2))
```

specifies 2 text strings to appear at line 6, position 2 and line 7, position 2 respectively.

Portability Considerations In Visual LANSAs this parameter should only be edited using the screen or report painter which will replace any text with a text map. DO NOT enter text using the command prompt or free format editor as it will not pass the full function checker if checked in to LANSAs for i.

All Platforms

The text map is used by the screen or report design facilities to store the details of all the text strings associated with the screen panel or report lines.

Once a screen or report layout has been "painted" and saved, all text details from the layout are stored in a "text map". The text map is then subsequently changed by using the "painter" again.

The presence of a text map is indicated by a TEXT parameter that looks like this example:

```
TEXT((*TMAPnnn 1 1))
```

where "nnn" is a unique number (within this function) that identifies the stored text map.

Some **very important** things about "text maps" and *TMAPnnn identifiers that you **must** know are:

- Never specify *TMAPnnn identifiers of your own or change *TMAPnnn identifiers to other values. Leave the assignment and management of *TMAPnnn identifiers to the screen and report design facilities.
- When copying a command that has an *TMAPnnn identifier, remove the *TMAPnnn references (ie: the whole TEXT parameter) from the copied command. If you fail to do this, then the full function checker will detect the

duplicated use of *TMAPnnn identifiers, and issue a fatal error message before any loss occurs.

- Never remove an *TMAPnnn identifier from a command. If this is done then the associated text map may be deleted, or reused in another command, during a full function check or compilation. Loss of text details is likely to result.
- Never "comment out" a command that contains a valid *TMAPnnn identifier. This is just another variation of the preceding warning and it runs the same risks of loss or reuse of text.
- Never specify *TMAPnnn values in an Application Template. In the template context *TMAPnnn values have no meaning. Use the "text string" format in commands used in, and initially generated by, Application Templates.

CURSOR_LOC

Specifies any user controlled cursor positioning that is required. The CURSOR_LOC parameter must always contain 2 values which may take any of the following forms:

*NONE / *NONE: which are the default values indicate that no user controlled cursor positioning is required. Normal LANSAs cursor control is to be used. When a screen is displayed the cursor will be positioned to either the first input capable field or the first field in error.

*ATFIELD / Field name: specifies that the cursor should be positioned to the named field. If the named field is not on the display or a field error exists, normal LANSAs cursor control will be used. Otherwise the cursor will be positioned to the nominated field.

Row value / Column value: specifies that the "values" nominated indicate the row and column number at which the cursor is to be positioned. The "values" nominated may be an alphanumeric literal (e.g.: 15) or the name of a field that contains the value (e.g.: #ROW). In all cases the value must be numeric. If the row or column values are invalid or a field error exists, normal LANSAs cursor control will be used. Otherwise the cursor will be positioned at the row and column specified.

When the row and column option is used **and** the row and column values are specified **as fields** (rather than numeric literals), the row and column number that the cursor was at when the command completed execution will be **returned in them**.

Note: The CURSOR_LOC does not behave in the same way on Windows as on

IBM i. On a Windows platform the value retrieved is the first position of the field the cursor is currently in.

The feature is a useful way of retrieving the location of the screen cursor at the time the command completed execution. In cases where you wish to **retrieve** the cursor location, but do not want to **specify it** before output to the screen, use coding like this:

```
CHANGE FIELD(#ROW #COL) TO(0)
DISPLAY FIELDS(#FIELD1 .. #FIELD10) CURSOR_LOC(#ROW #COL)
```

When the DISPLAY command is executed #ROW and #COL are both zero, which is an invalid cursor location. In such cases normal LANSA cursor control is resumed and the user positioning request is ignored. However, after completion of the command fields #ROW and #COL will contain the location of the cursor at the time the DISPLAY command completed execution.

STD_HEAD

Specifies whether or not the standard LANSA design for the screen heading lines (lines 1 and 2) should be used.

*DFT, which is the default value, indicates that the system default value for the STD_HEAD parameter should be used. The system default value is stored in the LANSA system definition block. Refer to [The System Definition Data Areas](#) in the *LANSA for i User Guide* for more details of the system definition block and how to change it.

This default value is affected when used with the OPTIONS(*OVERLAY) parameter. Refer to the OPTIONS parameter for more details.

*YES indicates that the standard LANSA screen heading lines should be used. When this option is used lines 1 and 2 of the display are not available for the positioning of user fields.

*NO indicates that the standard LANSA screen heading lines should not be used. In this case lines 1 and 2 of the display can be used to position user fields.

OPTIONS

Specifies special display options for this screen panel.

*NONE, which is the default value, indicates that there are no special display options for this screen panel.

Otherwise, specify one or more of the following:

*NOREAD indicates that the details being displayed are **not to be** read back

from the screen. Thus the details are presented to the user, but cannot ever be read back into the program. Additionally, the program **does not stop** at the command and wait for a user interaction. The stop and wait event will only occur when a subsequent DISPLAY or REQUEST command is executed that does not use the *NOREAD option.

*OVERLAY indicates that the screen panel should overlay whatever details are already on the screen. **Details already on the screen** will become protected and can no longer be read from the device, but they will be visible to the user.

When *OVERLAY is used, the **default for the STD_HEAD** parameter is *NO. Therefore, unless STD_HEAD(*YES) is coded, the screen heading lines will not be displayed when using OPTIONS(*OVERLAY). Note that when a "standard heading" (*YES) is sent to the screen it causes the entire screen to be cleared. If STD_HEAD(*NO) is used it has no effect upon standard headings already on the screen from previous commands.

If either the *NOREAD or *OVERLAY options are used, the complete screen details **must fit on one screen panel**.

Note: These display options have been provided to allow emulation of IBM i 3GL programs, and will not be portable to other platforms. They are not supported by the current GUI or by LANSA for the Web. use of these options is therefore not recommended.

Portability This parameter is not supported in Visual LANSA
Considerations applications and should not be used. If used, a Full Function Check fatal error will be issued.

IGCCNV_KEY

Controls the appearance of the text "Fnn=XXXXXX" in the function key area, of the function key assigned to support IGC conversion.

This parameter is **ignored** if the language under which this function is being compiled does not have the "IGCCNV required" flag enabled, or if this function uses the *NOIGCCNV options keyword (refer to the FUNCTION command).

Also note that this parameter only controls the appearance of the text "Fnn=XXXXXX" in the function key area. It does **not** control the enablement of the IGCCNV DDS keyword in the display file associated with this function. This is controlled by the setting of the "IGCCNV required" flag and the use of the *NOIGCCNV option.

*AUTO, which is the default value, indicates that appearance of the function key text should be determined automatically. The automatic rules used to

determine whether or not to show the function key text are:

- If there are no fields with keyboard shift J, E or O involved, the text will not appear (ignore all following rules).
- For a REQUEST command the text will always appear.
- For DISPLAY or POP_UP commands, the current "mode" is tested. If the mode is "change" (ie: fields on the screen are input capable), the text will appear. For all other modes the text will not appear.

Other allowable values for this parameter are *YES, indicating that the text should always appear, or, *NO indicating that the text should never appear.

The final option allows the nomination of a condition previously defined by a DEF_COND command. If the condition is true the text should appear. If the condition is false, the text should not appear.

Portability The parameter is ignored in Visual LANSA applications with
Considerations no known effect to the application.

7.27.2 DISPLAY Comments / Warnings

The DISPLAY command is a "mode sensitive" command. For details of mode sensitive command processing, refer to [Screen Modes and Mode Sensitive Commands](#).

- The following table indicates all combinations of the DESIGN and IDENTIFY parameters and what values actually result when any of the default values are used:

Specified: DESIGN	Specified: IDENTIFY	LANSA uses: DESIGN	LANSA uses: IDENTIFY
*IDENTIFY	*DESIGN	*DOWN	*LABEL
*IDENTIFY	*COLHDG	*ACROSS	*COLHDG
*IDENTIFY	*LABEL	*DOWN	*LABEL
*IDENTIFY	*DESC	*DOWN	*DESC
*IDENTIFY	*NOID	*ACROSS	*NOID
*DOWN	*DESIGN	*DOWN	*LABEL
*DOWN	*COLHDG	*DOWN	*COLHDG
*DOWN	*LABEL	*DOWN	*LABEL
*DOWN	*DESC	*DOWN	*DESC
*DOWN	*NOID	*DOWN	*NOID
*ACROSS	*DESIGN	*ACROSS	*COLHDG
*ACROSS	*COLHDG	*ACROSS	*COLHDG
*ACROSS	*LABEL	*ACROSS	*LABEL
*ACROSS	*DESC	*ACROSS	*DESC
*ACROSS	*NOID	*ACROSS	*NOID

- The following table indicates all combinations of the DESIGN and

IDENTIFY parameters and what values result when the *DESIGN default is used in the associated DOWN_SEP or ACROSS_SEP parameters:

Specified: DESIGN	Specified: IDENTIFY	*DESIGN Specified: DOWN_SEP	*DESIGN Specified: ACROSS_SEP
*IDENTIFY	*DESIGN	1	1
*IDENTIFY	*COLHDG	5	1
*IDENTIFY	*LABEL	1	1
*IDENTIFY	*DESC	1	1
*IDENTIFY	*NOID	1	1
*DOWN	*DESIGN	1	1
*DOWN	*COLHDG	5	1
*DOWN	*LABEL	1	1
*DOWN	*DESC	1	1
*DOWN	*NOID	1	1
*ACROSS	*DESIGN	5	1
*ACROSS	*COLHDG	5	1
*ACROSS	*LABEL	1	1
*ACROSS	*DESC	1	1
*ACROSS	*NOID	1	1

- In some cases all the fields specified in the FIELDS parameter will not fit on one screen. In this case a second, third, fourth, etc. screen is automatically designed as required.
- In terms of the RDML program they can be treated like a single "long" screen. LANSAs will automatically process the screens one after another until they have all been processed. When all screens have been processed the next RDML command is executed. So when you use the DISPLAY command you may in fact be requesting that 2 or 3 or more screens be displayed one after

another.

- This facility is a feature of the automatic design procedures. If you are coding the RDML program yourself it may be advisable in some circumstances to "split up" the DISPLAY command into multiple DISPLAY commands that have only one screen format each.
- If you use an expandable group expression in a DISPLAY command FIELDS parameter and you change the layout using the report design facility, LANSAs will substitute the expression with the actual fields. This is the only way LANSAs can assign attributes to the individual fields, regardless of which group they initially came from.

7.27.3 DISPLAY Examples

Example 1: Display fields #ORDNUM, #CUSTNUM and #DATEDUE to the user.

```
DISPLAY FIELDS(#ORDNUM #CUSTNUM #DATEDUE)
```

or, identically:

```
GROUP_BY NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #  
DISPLAY FIELDS(#ORDERHEAD)
```

both use default values for all parameters and field attributes and thus would cause a screen something like this to be designed automatically:

```
Order number : 99999999  
Customer no  : 999999  
Date due    : 99/99/99
```

Example 2: Modify the previous example to design the screen across ways and use column headings to identify the fields:

```
GROUP_BY NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #  
DISPLAY FIELDS(#ORDERHEAD) DESIGN(*ACROSS) IDENTIFY(*CC
```

which would cause a screen something like this to be designed automatically:

```
Company      Order      Date  
Order        Customer  Order  
Number       Number    Due  
99999999    999999    99/99/99
```

Example 3: Display #ORDNUM #CUSTNUM and #DATEDUE and also specify specific positions and identification methods as field attributes.

For details of field attributes, refer to [Field Attributes and their Use](#).

When specific positions for a field are nominated the automatic design facility is effectively "disabled".

```
GROUP_BY NAME(#ORDERHEAD) FIELDS((#ORDNUM *COLHDG *L
```

```
DISPLAY FIELDS(#ORDERHEAD) DESIGN(*ACROSS) TEXT(('--
DATE--' 6 37) ('-----' 8 37))
```

which would cause a screen something like this to be designed:

```
Company          Customer no : 999999
Order
Number
99999999          --DATE--
                   99/99/99
                   -----
```

Note: The manual specification of row and column numbers and "text" is not required. The screen design facility can be used to modify an "automatic" design much more quickly and easily. Refer to [The Screen Design Facility](#) in the *LANSA for i User Guide* for details of how to use the screen design facility.

After the screen design facility has been used on a DISPLAY command the associated FIELDS parameter (in the DISPLAY or GROUP_BY command) will be automatically re-written with the required row, column and method of identification attributes. Remember, if an expandable group expression was used, LANSA will substitute the expression with the fields that constitute it.

In addition the TEXT parameter of the DISPLAY command will also be automatically re-written.

Example 4: Display the order header details used in the previous example and all the associated invoice lines nominated in a list named #ORDERLINE:

```
GROUP_BY  NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
DISPLAY   FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)
```

Since default values were used for all parameters and no field attributes were specified a screen something like this would be designed automatically:

```
Order number : 99999999
Customer no  : 999999
Date due    : 99/99/99
```

```

Line
No  Product Quantity Price
99  9999999 99999 99999.99
99  9999999 99999 99999.99
99  9999999 99999 99999.99
99  9999999 99999 99999.99
99  9999999 99999 99999.99
99  9999999 99999 99999.99

```

Example 5: Display the order header details used in the previous example and all the associated invoice lines nominated in a list named #ORDERLINE which only has 4 entries. Display invoice line totals (which can be adjusted) below the invoice lines:

```

GROUP_BY  NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
DISPLAY   FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE 4) OP
REQUEST   FIELDS(#TOTQTY #TOTPRICE) OPTIONS(*OVERLAY)

```

after screen painting to adjust the field positions to avoid overlapping, the resulting screen (after executing the above code) would look something like this:

```

Order number : 99999999
Customer no  : 999999
Date due    : 99/99/99

```

```

Line
No  Product Quantity Price
99  9999999 99999 99999.99
99  9999999 99999 99999.99
99  9999999 99999 99999.99
99  9999999 99999 99999.99

```

```

Total Qty   : 9999999
Total Price : 9999999.99

```

Example 6 : Use an expandable group expression and redesign the layout using the screen design facility:

```
GROUP_BY NAME(#XG_ORDHDG) FIELDS(#ORDNUM #CUSTNUM #
DISPLAY FIELDS(#XG_ORDHDG) DESIGN(*ACROSS) IDENTIFY(*CC
```

The screen designed automatically would look like this:

```
Company Order Date
Order Customer Order
Number Number Due
99999999 999999 99/99/99
```

If the layout is changed using the screen design facility to look like this:

```
Company Order
Order Customer
Number Number
99999999 999999 Date Order Due : 99/99/99
```

then the DISPLAY command FIELDS parameter will be expanded as follows:

```
DISPLAY FIELDS((#ORDNUM *L2 *P3) (#CUSTNUM *L2 *P37) (#DAI
```

7.28 DLT_ENTRY

The DLT_ENTRY command is used to delete an individual entry from a list. The list specified must be a working list (used to store information within a program). It is not possible to use the DLT_ENTRY command against a browse list (used for displaying information at a workstation).

Refer to the DEF_LIST command for more details of lists and list processing.

Also See

[7.28.1 DLT_ENTRY Parameters](#)

[7.28.2 DLT_ENTRY Comments / Warnings](#)

[7.28.3 DLT_ENTRY Examples](#)

Optional

```
DLT_ENTRY ---- NUMBER ----- *CURRENT -----  
----->
```

numeric value or field name

```
>-- FROM_LIST ---- *FIRST -----|  
list name
```

7.28.1 DLT_ENTRY Parameters

FROM_LIST

NUMBER

NUMBER

Specifies the list entry number that is to be deleted.

The default value of *CURRENT specifies that the entry currently selected (ie retrieved) from the list, in a SELECTLIST/ENDSELECT list processing loop or by a GET_ENTRY or LOC_ENTRY command, will be deleted from the list.

A numeric value or field name specifies the entry number of the list that is to be deleted. As each entry is added to a list by the ADD_ENTRY command it is assigned a number that identifies it. List entries are numbered from 1 (first entry number) to 9999 (maximum possible last entry number). This entry number can then vary as ADD_ENTRY commands, to add after previous list entries, or DLT_ENTRY commands, to delete previous list entries, are executed. By specifying a list entry number it is possible to delete an individual list entry without first having selected (or retrieved) it.

FROM_LIST

Specifies the name of the list from which the entry should be deleted.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which must be a working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

7.28.2 DLT_ENTRY Comments / Warnings

- Use of the DLT_ENTRY command may incur a performance penalty because of the underlying implementation of working lists. Heavy use should be benchmarked with realistically sized data sets before being put into a production environment. Possible design alternatives include replacement of the working list by a keyed work file and construction of a second working list from the first.
- Use of the DLT_ENTRY command also means that the entry number of all entries succeeding the deleted entry are reduced by 1. This may cause problems where the entry number of a particular entry is assumed to remain static, for example where "pointers" to working list entries are used and also where the entries in a list are processed in a loop other than SELECTLIST/ENDSELECT.

7.28.3 DLT_ENTRY Examples

Example 1: Delete the 3rd entry from the order line working list, which is the first list defined in the program:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
*   ... <entries added to the order line list via ADD_ENTRY>
GET_ENTRY NUMBER(3) FROM_LIST(#ORDERLINE)
DLT_ENTRY
```

which is equivalent to:

```
GET_ENTRY NUMBER(3) FROM_LIST(#ORDERLINE)
DLT_ENTRY NUMBER(*CURRENT) FROM_LIST(*FIRST)
```

which is also equivalent to:

```
GET_ENTRY NUMBER(3) FROM_LIST(#ORDERLINE)
DLT_ENTRY NUMBER(*CURRENT) FROM_LIST(#ORDERLINE)
```

which is also equivalent to:

```
DLT_ENTRY NUMBER(3) FROM_LIST(#ORDERLINE)
```

Example 2: Delete all entries in an existing working list named #ORDERLINE, where the field #QUANTITY is less than or equal to 0:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
DEFINE FIELD(#ENTRY) TYPE(*DEC) LENGTH(7) DECIMALS(0)

SELECTLIST NAMED(#ORDERLINE)
IF COND('#QUANTITY *LE 0')
DLT_ENTRY NUMBER(*CURRENT) FROM_LIST(#ORDERLINE)
ENDIF
ENDSELECT
```

7.29 DLT_LIST

The DLT_LIST command is used to delete all entries in a list.

The list may be a **browse** list (used for displaying information at a workstation) or a **working** list (used to store information within a program).

Normally it is only ever used when exiting from a function or to reduce the number of "active" browse lists in a function. Only 10 browse lists can be "active" (ie: contain information) at any one time within a function.

Executing a DLT_LIST command against a working list is functionally identical to executing a CLR_LIST command.

Deleting a list that contains no entries is a valid operation. No error will result.

Also See

[7.29.1 DLT_LIST Parameters](#)

[7.29.2 DLT_LIST Examples](#)

Optional

```
DLT_LIST ----- NAMED ----- *FIRST -----  
---|  
           name of list
```

7.29.1 DLT_LIST Parameters

NAMED

NAMED

Specifies the name of the list which is to be deleted.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which may be a browse or a working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

Refer to the DEF_LIST command for more details of lists and their uses.

7.29.2 DLT_LIST Examples

This example applies to the DLT_LIST command.

Delete a list named #ORDERLINE:

```
DLT_LIST NAMED(#ORDERLINE)
```

7.30 DOUNTIL

The DOUNTIL command is used to create a conditional loop and to continue looping until the condition is true. The DOUNTIL command loop is "delimited" by the associated ENDUNTIL command which must be present.

Note: The DOUNTIL command is similar in structure to the DOWHILE command. However, there is one important difference. In the DOUNTIL command, the condition is not checked BEFORE doing the first iteration.

Refer to the ENDUNTIL command for more information and examples of both of these commands.

Also See

[7.30.1 DOUNTIL Parameters](#)

[7.30.2 DOUNTIL Examples](#)

[7.3 BEGIN_LOOP](#)

[7.33 END_LOOP](#)

[7.40 ENDUNTIL](#)

[7.31 DOWHILE](#)

[7.41 ENDWHILE](#)

Required

```
DOUNTIL ----- COND -----'condition'-----  
--|
```

7.30.1 DOUNTIL Parameters

COND

COND

Specifies the condition which is to be met to cause the loop to terminate. For details of how conditions and expressions are specified in LANSAs, refer to [Specifying Conditions and Expressions](#).

7.30.2 DOUNTIL Examples

Executing a DOUNTIL . . . ENDUNTIL Routine

Executing DOUNTIL . . . ENDUNTIL to enter "n" records to a file

Using DEF_COND Values as DOUNTIL parameters to make code easier to read and maintain

Comparing DOUNTIL . . . ENDUNTIL to the use of IF . . . GOTO . . . ENDIF

Executing DOUNTIL . . . ENDUNTIL routine with an Array Index

Executing a DOUNTIL . . . ENDUNTIL Routine

This is an example of the simple use of DOUNTIL and ENDUNTIL to count to 10 in a loop:

```
DEFINE  FIELD(#COUNT) REFFLD(#STD_NUM)
CHANGE  FIELD(#COUNT) TO(1)
DOUNTIL COND('#COUNT > 10')
DISPLAY FIELDS(#COUNT)
CHANGE  FIELD(#COUNT) TO('#COUNT + 1')
ENDUNTIL
```

The DOUNTIL command is similar in structure to the DOWHILE command. However, there is one important difference. In the DOUNTIL command, the condition is not checked BEFORE doing the first iteration. This is an example:

```
DEFINE  FIELD(#COUNT) REFFLD(#STD_NUM)
CHANGE  FIELD(#COUNT) TO(5)
DOUNTIL COND('#COUNT > 1')
DISPLAY FIELDS(#COUNT)
ENDUNTIL
```

Although #COUNT is greater than 1, the loop is still executed one time.

Executing DOUNTIL . . . ENDUNTIL to enter "n" records to a file

In this example, the details of 10 employees are inserted into a file:

```
GROUP_BY NAME(#EMPDET) FIELDS(#EMPNO #SURNAME #SALARY)
DEFINE  FIELD(#COUNT) REFFLD(#STD_NUM)
DEF_LIST NAME(#WORKER) FIELDS(#EMPNO #SURNAME #SALARY)
CHANGE  FIELD(#COUNT) TO(1)
DOUNTIL COND('#COUNT > 10')
```

```

DISPLAY  FIELDS(#COUNT)
REQUEST  FIELDS(#EMPNO #SURNAME #SALARY)
ADD_ENTRY TO_LIST(#WORKER)
CHANGE   FIELD(#COUNT) TO('#COUNT + 1')
ENDUNTIL
DISPLAY  BROWSELIST(#WORKER)

```

Using DEF_COND Values as DOUNTIL parameters to make code easier to read and maintain

In this example, the COND parameter for the DOUNTIL command is set by the DEF_COND command before DOUNTIL is executed.

```

DEFINE   FIELD(#COUNT) REFFLD(#STD_NUM)
DEF_COND NAME(*COUNT_TEN) COND('#COUNT > 10')
CHANGE   FIELD(#COUNT) TO(1)
DOUNTIL  COND(*COUNT_TEN)
DISPLAY  FIELDS(#COUNT)
CHANGE   FIELD(#COUNT) TO('#COUNT + 1')
ENDUNTIL

```

The use of DEF_COND allows the programmer to give a complex condition a meaningful label that expresses the reason behind the test of the condition. When subsequent programmers read the DOWHILE statement, the meaningful label will help them to understand the purpose of the statement.

The use of DEF_COND also helps in situations where the same condition is referred to multiple times in the function. In this case it reduces the quantity of code and makes maintenance of the condition easier. . For further reference, refer to the << link to DEF_COND >> command.

Comparing DOUNTIL . . . ENDUNTIL to the use of IF . . . GOTO . . . ENDIF

This example, shows the simple use of the DOUNTIL ENDUNTIL routine:

```

DOUNTIL  COND('#A >= B')
    << logic >>
    << logic >>
    << logic >>
ENDUNTIL

```

Now compare this to the use of the IF GOTO ENDIF routine:

```
L01: IF      COND('#A < B')
      << logic >>
      << logic >>
      << logic >>
GOTO  LABEL(L01)
ENDIF
```

When compared, the use of the DOUNTIL ENDUNTIL routine is simpler and easier to read than when the IF GOTO ENDIF routine is used for a simple loop.

Executing DOUNTIL . . . ENDUNTIL routine with an Array Index

This example demonstrates the use of the DOUNTIL ENDUNTIL routine with an Array Index that groups 3 field values into an array, increments each one by 10%, then adds the resulting values up to display:

```
DEFINE  FIELD(#VAL1) REFFLD(#STD_NUM)
DEFINE  FIELD(#VAL2) REFFLD(#STD_NUM)
DEFINE  FIELD(#VAL3) REFFLD(#STD_NUM)
DEFINE  FIELD(#I1) REFFLD(#STD_NUM)
DEFINE  FIELD(#TOTAL) TYPE(*DEC) LENGTH(6) DECIMALS(2) LA
DEF_ARRAY NAME(#ARR) INDEXES(#I1) OF_FIELDS(#VAL1 #VAL2 #
CHANGE  FIELD(#TOTAL) TO(1)
CHANGE  FIELD(#I1) TO(1)
REQUEST FIELDS(#VAL1 #VAL2 #VAL3)
DOUNTIL COND('#I1 > 3')
CHANGE  FIELD(#ARR#I1) TO('#ARR#I1 * 1.1')
CHANGE  FIELD(#TOTAL) TO('#TOTAL + #ARR#I1')
CHANGE  FIELD(#I1) TO('#I1 + 1')
ENDUNTIL
DISPLAY FIELDS(#TOTAL)
```

Refer to the [7.30 DOUNTIL](#) command for further reference to the array index.

7.31 DOWHILE

The DOWHILE command is used to create a conditional loop and to continue looping whilst the condition is true. The end of the loop is "delimited" by the associated ENDWHILE command.

Note: The DOUNTIL command is similar in structure to the DOWHILE command. However, in the DOUNTIL command, the condition is not checked BEFORE doing the first iteration.

Refer to the ENDWHILE command for more information and examples of both of these commands.

Also See

[7.31.1 DOWHILE Parameters](#)

[7.31.2 DOWHILE Examples](#)

[7.3 BEGIN_LOOP](#)

[7.30 DOUNTIL](#)

[7.33 END_LOOP](#)

[7.40 ENDUNTIL](#)

[7.41 ENDWHILE](#)

Required

```
DOWHILE ----- COND ----- 'condition' -----  
--|
```

7.31.1 DOWHILE Parameters

COND

COND

Specifies the condition which is to be met to continue the loop processing. For details of how conditions and expressions are specified using LANSAs, refer to [Specifying Conditions and Expressions](#).

7.31.2 DOWHILE Examples

Executing a DOWHILE . . . ENDWHILE Routine

Executing DOWHILE . . . ENDWHILE to enter "n" records to a file

Comparing DOWHILE . . . ENDWHILE to the use of IF . . . GOTO . . . ENDIF

Executing DOWHILE . . . ENDWHILE routine with an Array Index

Executing a DOWHILE . . . ENDWHILE Routine

This is an example of the simple use of DOWHILE and ENDWHILE to count to 10 in a loop:

```
DEFINE  FIELD(#COUNT) REFFLD(#STD_NUM)
CHANGE  FIELD(#COUNT) TO(1)
DOWHILE COND('#COUNT <= 10')
DISPLAY FIELDS(#COUNT)
CHANGE  FIELD(#COUNT) TO('#COUNT + 1')
ENDWHILE
```

The DOWHILE command is similar in structure to the DOUNTIL command. However, there is one important difference. In the DOWHILE command, the condition is checked BEFORE doing the first iteration.

Executing DOWHILE . . . ENDWHILE to enter "n" records to a file

In this example, the details of 10 employees are inserted into a file:

```
GROUP_BY  NAME(#EMPDET) FIELDS(#EMPNO #SURNAME #SALARY)
DEFINE  FIELD(#COUNT) REFFLD(#STD_NUM)
DEF_LIST NAME(#WORKER) FIELDS(#EMPNO #SURNAME #SALARY)
CHANGE  FIELD(#COUNT) TO(1)
DOWHILE COND('#COUNT <= 10')
DISPLAY FIELDS(#COUNT)
REQUEST  FIELDS(#EMPNO #SURNAME #SALARY)
ADD_ENTRY TO_LIST(#WORKER)
CHANGE  FIELD(#COUNT) TO('#COUNT + 1')
ENDWHILE
DISPLAY  BROWSELIST(#WORKER)
```

Using DEF_COND Values as DOWHILE parameters to make code easier to

read and maintain

In this example, the COND parameter for the DOWHILE command is set by the DEF_COND command before DOWHILE is executed.

```
DEFINE  FIELD(#COUNT) REFFLD(#STD_NUM)
DEF_COND  NAME(*COUNT_TEN) COND('#COUNT <= 10')
CHANGE  FIELD(#COUNT) TO(1)
DOWHILE  COND(*COUNT_TEN)
DISPLAY  FIELDS(#COUNT)
CHANGE  FIELD(#COUNT) TO('#COUNT + 1')
ENDWHILE
```

The use of DEF_COND allows the programmer to give a complex condition a meaningful label that expresses the reason behind the test of the condition. When subsequent programmers read the DOWHILE statement, the meaningful label will help them to understand the purpose of the statement.

The use of DEF_COND also helps in situations where the same condition is referred to multiple times in the function. In this case it reduces the quantity of code and makes maintenance of the condition easier. For further details, refer to the [DEF_COND](#) command.

Comparing DOWHILE . . . ENDWHILE to the use of IF . . . GOTO . . . ENDIF

In this example, we see the simple use of the DOWHILE ENDWHILE routine:

```
DOWHILE  COND('#A < B')
    << logic >>
    << logic >>
    << logic >>
ENDWHILE
```

Now compare this to the use of the IF GOTO ENDIF routine:

```
L01: IF    COND('#A < B')
    << logic >>
    << logic >>
    << logic >>
GOTO    LABEL(L01)
```

ENDIF

When compared, the use of the DOWHILE ENDWHILE routine is simpler and easier to read than when the IF GOTO ENDIF routine is used for a simple loop.

Executing DOWHILE . . . ENDWHILE routine with an Array Index

This example demonstrates the use of the DOWHILE ENDWHILE routine with an Array Index that groups 3 field values into an array, increments each one by 10%, then adds the resulting values up to display:

```
DEFINE FIELD(#VAL1) REFFLD(#STD_NUM)
DEFINE FIELD(#VAL2) REFFLD(#STD_NUM)
DEFINE FIELD(#VAL3) REFFLD(#STD_NUM)
DEFINE FIELD(#I1) REFFLD(#STD_NUM)
DEFINE FIELD(#TOTAL) TYPE(*DEC) LENGTH(6) DECIMALS(2) LABEL
DEF_ARRAY NAME(#ARR) INDEXES(#I1) OF_FIELDS(#VAL1 #VAL2 #VAL3)
CHANGE FIELD(#TOTAL) TO(1)
CHANGE FIELD(#I1) TO(1)
REQUEST FIELDS(#VAL1 #VAL2 #VAL3)
DOWHILE COND('#I1 <= 3')
CHANGE FIELD(#ARR#I1) TO('#ARR#I1 * 1.1')
CHANGE FIELD(#TOTAL) TO('#TOTAL + #ARR#I1')
CHANGE FIELD(#I1) TO('#I1 + 1')
ENDWHILE
DISPLAY FIELDS(#TOTAL)
```

Refer to the [DEF_ARRAY](#) command for further reference to the array index.

Refer to the [7.17 DEF_ARRAY](#) command for further reference to the array index.

7.32 ELSE

The ELSE command is used in conjunction with the IF command and specifies what is to happen if the IF condition is not true.

Refer to the IF and ENDIF commands for more details and examples of these commands.

Also See

[7.32.2 ELSE Examples](#)

[7.32.1 ELSE Parameters](#)

ELSE ----- no parameters -----|

7.32.1 ELSE Parameters

There are no ELSE Parameters.

7.32.2 ELSE Examples

Example 1: If field #I is greater than 10 issue a message indicating this, else issue a message indicating it is less than or equal to 10:

```
IF    COND('#I *GT 10')
MESSAGE MSGTXT('#I is greater than 10')
ELSE
MESSAGE MSGTXT('#I is less than or equal to 10')
ENDIF
```

Example 2: Execute a certain series of commands if #QUANTITY is less than 10 and #MEASURE is greater than 42.67, else execute a different series of commands:

```
IF    COND('( #QUANTITY *LT 10) *AND (#MEASURE *GT 42.67)')
* << commands to execute when condition is true >>
ELSE
* << commands to execute when condition is false >>
ENDIF
```

7.33 END_LOOP

The END_LOOP command is used to delimit a processing loop that was started by a BEGIN_LOOP command.

Also See

[7.33.1 END_LOOP Parameters](#)

[7.33.2 END_LOOP Examples](#)

END_LOOP ----- no parameters -----
--|

7.33.1 END_LOOP Parameters

There are no END_LOOP parameters

7.33.2 END_LOOP Examples

Refer to the [7.3.2 BEGIN_LOOP Examples](#).

7.34 ENDCASE

The ENDCASE command is used in conjunction with the CASE command and specifies the end of a case block of statements.

Refer to the CASE, WHEN and OTHERWISE commands for more details and examples of these commands.

Also See

[7.34.1 ENDCASE Parameters](#)

[7.34.2 ENDCASE Examples](#)

[7.8 CASE](#)

[7.73 OTHERWISE](#)

[7.100 WHEN](#)

ENDCASE ----- no parameters -----
-|

7.34.1 ENDCASE Parameters

There are no ENDCASE parameters.

7.34.2 ENDCASE Examples

Refer to [7.8.3 CASE Examples](#) for use of the ENDCASE Command

7.35 ENDCHECK

The ENDCHECK command is used in conjunction with the BEGNCHECK command and identifies the end of a block of validation checks.

In addition it also indicates what is to happen if a validation error was detected within the validation block.

Refer to the BEGNCHECK command for more details and examples of this command.

Also See

[7.35.1 ENDCHECK Parameters](#)

[7.35.2 ENDCHECK Comments / Warnings](#)

[7.35.3 ENDCHECK Examples](#)

Optional

```
ENDCHECK ----- IF_ERROR ----- *LASTDIS -----
----->
        *NEXT
        *RETURN
        label

>-- MSGTXT ----- *NONE ----->
        message text

>-- MSGID ----- *NONE ----->
        message identifier

>-- MSGF ----- DC@M01 . *LIBL -----
>
        message file . library name

>-- MSGDTA ----- substitution variables -----|
        |expandable group expression|
        ----- 20 max -----
```

7.35.1 ENDCHECK Parameters

IF_ERROR

MSGDTA

MSGF

MSGID

MSGTXT

IF_ERROR

Specifies what action is to be taken if one or more of the validation commands inside the validation block find a validation error. The validation check commands used inside a validation block include CONDCHECK, RANGECHECK, VALUECHECK, CALLCHECK and DATECHECK.

A validation error occurs within a validation block when one or more of the validation commands detects a condition that has an "action" of *ERROR specified. Refer to the validation commands for more details of how and when the *ERROR "action" is specified in a validation command.

In addition, a validation error can also be caused by the execution of a SET_ERROR command within the validation block. This usually occurs when user written validation logic is being used rather than one of the 6 standard validation check commands.

Finally, a validation error can be caused by the execution (within the validation block) of any database I/O command that receives a "VE" (validation error) return code. Validation errors are normally only received by the INSERT, UPDATE and DELETE database I/O commands.

*LASTDIS, which is the default value, specifies that control should be passed back to the last display screen used. The field(s) that failed the validation check(s) within the validation block or had a SET_ERROR command executed against them will be displayed in reverse image and the cursor positioned to the first field in error on the screen.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

Note that *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as

"&1 are out of stock ... reorder &2"

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA(''BOLTS  '' #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.35.2 ENDCHECK Comments / Warnings

BEGINCHECK / ENDCHECK blocks can be nested. However, if an "inner" block detects an error it also triggers an error in all associated "outer" blocks.

This can be illustrated like this:

```
BEGINCHECK
  BEGINCHECK
    BEGINCHECK
      A validation error in this block will "trigger" a
      validation error at all levels (marked by <-).
    ENDCHECK <-
  ENDCHECK <-
ENDCHECK <-
```

The ability to nest BEGINCHECK / ENDCHECK commands is particularly useful when processing screens that have browse lists that are used for data entry. Consider a data entry screen like this:

```
Order number : 99999999
Customer no  : 999999
Date due    : 99/99/99
```

```
Line
No  Product Quantity Price
99  9999999  99999  99999.99
99  9999999  99999  99999.99
99  9999999  99999  99999.99
99  9999999  99999  99999.99
99  9999999  99999  99999.99
99  9999999  99999  99999.99
```

The RDML program to process data entered in this way might look something like this:

```
GROUP_BY  NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU
```

```

SET_MODE TO(*ADD)
INZ_LIST NAMED(#ORDERLINE) NUM_ENTRYS(20)
L1: REQUEST FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)

BEGINCHECK

  << validate order header details >>

SELECTLIST NAMED(#ORDERLINE) GET_ENTRYS(*NOTNULL) <-
  BEGINCHECK
  << validate order line details >>
  IF_ERROR
  UPD_ENTRY IN_LIST(#ORDERLINE)
  ENDIF
  ENDCHECK IF_ERROR(*NEXT)
ENDSELECT -----

  ENDCHECK IF_ERROR(*LASTDIS)
  << update database >>

```

Note that the "inner" BEGINCHECK/ENDCHECK loop is processed for each browse list entry that the user entered. Note also that the IF_ERROR(*NEXT) parameter causes the SELECTLIST loop to continue to process all browse list entries and not stop the first time an error is detected.

The "outer" BEGINCHECK/ENDCHECK command uses the IF_ERROR(*LASTDIS) parameter which will cause the REQUEST command to be re-executed if a validation error is detected. A validation error will be "detected" if

- an error is found in the order header details.

or

- an error is found in **one or more** of the order line details. This happens because any error in the "inner" validation block also triggers an error in the "outer" validation block.

7.35.3 ENDCHECK Examples

Example 1: Request that the user input a product number, an order number and a quantity then perform validation checks against the fields:

```
REQUEST  FIELDS(#PRODNO #ORDNUM #QUANTITY)

BEGINCHECK
FILECHECK FIELD(#PRODNO) USING_FILE(PRODUCT) MSGTXT('Pr
RANGECHECK FIELD(#ORDNUM) RANGE(A000000 Z999999) MSGTX'
RANGECHECK FIELD(#QUANTITY) RANGE(1 9999) MSGTXT('Quantit
ENDCHECK
```

is identical to the following example, because of the default value IF_ERROR(*LASTDIS) on the ENDCHECK command:

```
REQUEST  FIELDS(#PRODNO #ORDNUM #QUANTITY)
L1:
BEGINCHECK
FILECHECK FIELD(#PRODNO) USING_FILE(PRODUCT) MSGTXT('Pr
RANGECHECK FIELD(#ORDNUM) RANGE(A000000 Z999999) MSGTX'
RANGECHECK FIELD(#QUANTITY) RANGE(1 9999) MSGTXT('Quantit
ENDCHECK  IF_ERROR(L1)
```

Example 2: The *LASTDIS default value actually means the "last display at this nesting level (or higher)" as is indicated in the following example:

```
REQUEST  FIELDS(#FIELD01) <-----
IF      COND('#FIELD01 *LT 10')          |
      REQUEST  FIELDS(#FIELD02) <----- |
      BEGINCHECK                          | |
      RANGECHECK FIELD(#FIELD01) RANGE(5 9) | |
      RANGECHECK FIELD(#FIELD02) RANGE(10 20) | |
      ENDCHECK ----- |
ELSE                                         |
      BEGINCHECK                          |
      RANGECHECK FIELD(#FIELD01) RANGE(15 19) |
      ENDCHECK -----
ENDIF
```

The arrows indicate where control is passed if either of the ENDCHECK commands find a validation error in their respective validation blocks.

If this example was coded it may be hard to use because the user does not get a chance to correct an error in FIELD01 if the value specified is less than 10 but not in the range 5 to 9. In this case the error message would appear on line 22/24 of the screen but the FIELD01 would not be on the display and thus could not be corrected.

This example might be better coded as:

```
REQUEST  FIELDS(#FIELD01) <-----
BEGINCHECK
IF      COND('#FIELD01 *LT 10')
RANGECHECK FIELD(#FIELD01) RANGE(5 9)
ELSE
RANGECHECK FIELD(#FIELD01) RANGE(15 19)
ENDIF
ENDCHECK -----

IF      COND('#FIELD01 *LT 10')
  REQUEST  FIELDS(#FIELD02) <-----
  BEGINCHECK
  RANGECHECK FIELD(#FIELD02) RANGE(10 20)
  ENDCHECK -----
ELSE
ENDIF
```

7.36 ENDIF

The ENDIF command is used in conjunction with the IF command and specifies the end of a IF condition.

Refer to the [7.53 IF](#) and [7.32 ELSE](#) commands for more details and examples of these commands.

Also See

[7.36.1 ENDIF Parameters](#)

[7.36.2 ENDIF Examples](#)

ENDIF ----- no parameters -----|

7.36.1 ENDIF Parameters

There are no ENDIF parameters.

7.36.2 ENDIF Examples

Refer to the [7.53.2 IF Examples](#) and [7.32.2 ELSE Examples](#) for examples of the ENDIF command.

7.37 ENDPRINT

The ENDPRINT command is used to end (close) a printer file that was previously opened by a PRINT or UPRINT command and cause a message to be issued that indicates how many pages were printed into the report.

If a printer file is opened by a PRINT or UPRINT command and no ENDPRINT command is present in the RDML program it will be automatically closed when the function ends.

If an ENDPRINT command is executed against a report that has never been opened it is ignored.

Portability Considerations When associated with a UPRINT command, a build warning will be generated if used in Visual LANSA code. An error will occur at execution time. Code using this facility can be conditioned so that it is not executed in this environment. Fully supported when associated with any other report command (e.g. PRINT, SKIP, SPACE).

Also See

[7.37.1 ENDPRINT Parameters](#)

[7.37.2 ENDPRINT Examples](#)

Optional

```
ENDPRINT ----- REPORT_NUM ---- 1 -----
----->
           report number 1 -> 8

>-- IO_STATUS ---- *STATUS ----->
           field name

>-- IO_ERROR ----- *ABORT -----|
           *NEXT
           *RETURN
           label
```

7.37.1 ENDPRINT Parameters

[IO_ERROR](#)

[IO_STATUS](#)

[REPORT_NUM](#)

REPORT_NUM

Specifies the number of the report that is to be ended. If no report number is specified report number 1 is assumed. The report number specified can be any number in the range 1 to 8.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation. This parameter is only valid for printer files opened by the UPRINT command.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

For values, refer to [I/O Return Codes](#) .

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed. This parameter is only valid for printer files opened by the UPRINT command.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command

label to which control should be passed.

7.37.2 ENDPRINT Examples

Example 1: Consider the 2 following RDML programs which ask a user to input an order number and then print the order line details:

```
GROUP_BY NAME(#ORDERDET) FIELDS(#ORDNUM #CUSTNUM #
REQUEST FIELDS(#ORDNUM)
L1: FETCH FIELDS(#ORDERDET) FROM_FILE(ORDHDR) WITH_KEY(
SELECT FIELDS(#ORDERDET) FROM_FILE(ORDLIN) WITH_KEY(
UPRINT FIELDS(#ORDERDET)
ENDSELECT
GOTO L1
```

and:

```
GROUP_BY NAME(#ORDERDET) FIELDS(#ORDNUM #CUSTNUM #

REQUEST FIELDS(#ORDNUM)
L1: FETCH FIELDS(#ORDERDET) FROM_FILE(ORDHDR) WITH_KEY(
SELECT FIELDS(#ORDERDET) FROM_FILE(ORDLIN) WITH_KEY(
UPRINT FIELDS(#ORDERDET)
ENDSELECT

ENDPRINT
GOTO L1
```

Note that both programs are almost identical. They both request that an order number be input, retrieve the order header details and then print all associated order line details.

The difference is in the ENDPRINT command.

The first program waits until the user uses the EXIT or MENU function key on the REQUEST command before an ENDPRINT command is automatically executed as the function ends. Thus **all** the orders requested have been printed into one print / spool file.

In the second program an ENDPRINT is executed after **each** order has been printed. This causes the print / spool file to be closed. A new one will be automatically opened when the next UPRINT command (if any) is executed. Thus each order is placed into a separate print / spool file.

7.38 ENDROUTINE

The ENDROUTINE command is used to indicate the end of:

- a subroutine (which starts with a SUBROUTINE command).
- an event routine (which starts with an EVTROUTINE command).
- a method routine (which starts with a MTHROUTINE command).
- a property routine (which starts with a PTYROUTINE command).

Every SUBROUTINE, EVTROUTINE, MTHROUTINE or PTYROUTINE command must have one and only one ENDROUTINE command.

When an ENDROUTINE command is executed control is returned to the logic that invoked the routine:

- Executing an ENDROUTINE command in a SUBROUTINE returns control to the command following the EXECUTE command that caused the subroutine to be executed.
- Executing an ENDROUTINE command in an EVTROUTINE, MTHROUTINE or PTYROUTINE returns control to the control logic that invoked the routine. The control logic will then decide what to do next based on the way in which the routine was invoked.

Portability Considerations Subroutines that are nested inside one another are not supported in the current release of Visual LANSA. This is a very rarely used coding technique and thus unlikely to cause any problems. In the event of problems simply un-nest the subroutine(s) involved and recompile.

Also See

[7.38.1 ENDROUTINE Parameters](#)

[7.38.2 ENDROUTINE Examples](#)

```
ENDROUTINE --- no parameters -----  
---|
```

7.38.1 ENDROUTINE Parameters

There are no ENDROUTINE parameters.

7.38.2 ENDROUTINE Examples

For examples of using this command, refer to:

[7.92.3 SUBROUTINE Examples - Part 1](#) and [7.92.4 SUBROUTINE Examples - Part 2](#)

[EVTROUTINE Examples](#)

[MTHROUTINE Examples](#), and

[PTYROUTINE Examples](#).

7.39 ENDSELECT

The ENDSELECT command has 2 uses:

- To "delimit" a database file processing loop that is caused by a SELECT command.
- To "delimit" a list processing loop that is caused by a SELECTLIST command.

Refer to the SELECT and SELECTLIST commands for more details and examples of this command.

Also See

[7.39.1 ENDSELECT Parameters](#)

[7.39.2 ENDSELECT Examples](#)

ENDSELECT ---- no parameters -----
--|

7.39.1 ENDSELECT Parameters

There are no ENDSELECT Parameters.

7.39.2 ENDSELECT Examples

Example 1: Select and print fields #ORDLIN, #PRODUCT, #QUANTITY and #PRICE from records in an order lines file which have an order number matching what is specified in field #ODRNUM:

```
SELECT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE) FROM_  
UPRINT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)  
ENDSELECT
```

or identically:

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU.  
SELECT  FIELDS(#ORDERLINE) FROM_FILE(ORDLIN) WITH_KEY(#(  
UPRINT  FIELDS(#ORDERLINE)  
ENDSELECT
```

Example 2: Select and print fields #ORDLIN, #PRODUCT, #QUANTITY and #PRICE from records in an order lines file which have a #QUANTITY value greater than 10 or a #PRICE value less than 49.99

```
SELECT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE) FROM_  
UPRINT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)  
ENDSELECT
```

or identically:

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU.  
SELECT  FIELDS(#ORDERLINE) FROM_FILE(ORDLIN) WHERE('( #QU  
UPRINT  FIELDS(#ORDERLINE)  
ENDSELECT
```

Example 3: Process all "altered" entries from a list named #ORDERLINE. Refer to the DEF_LIST command for more details of lists and list processing.

```
SELECTLIST NAMED(#ORDERLINE) GET_ENTRYS(*ALTERED)  
*    << Commands to process the list >>  
*    << Commands to process the list >>  
*    << Commands to process the list >>  
ENDSELECT
```

7.40 ENDUNTIL

The ENDUNTIL command is used to "delimit" the processing loop caused by a DOUNTIL command.

Note: The DOUNTIL command is similar in structure to the DOWHILE command. However, in the DOUNTIL command, the condition is not checked BEFORE doing the first iteration.

Also See

[7.40.1 ENDUNTIL Parameters](#)

[7.40.2 ENDUNTIL Examples](#)

[7.30 DOUNTIL](#)

[7.31 DOWHILE](#)

[7.41 ENDWHILE](#)

[7.33 END_LOOP](#)

[7.3 BEGIN_LOOP](#)

ENDUNTIL ----- no parameters -----
--|

7.40.1 ENDUNTIL Parameters

There are no ENDUNTIL Parameters.

7.40.2 ENDUNTIL Examples

Refer to the [7.30.2 DOUNTIL Examples](#) for examples of using this command.

7.41 ENDWHILE

The ENDWHILE command is used to "delimit" the processing loop caused by a DOWHILE command.

Also See

[7.41.1 ENDWHILE Parameters](#)

[7.41.2 ENDWHILE Examples](#)

[7.3 BEGIN_LOOP](#)

[7.30 DUNTIL](#)

[7.31 DOWHILE](#)

[7.33 END_LOOP](#)

[7.40 ENDUNTIL](#)

ENDWHILE ----- no parameters -----
--|

7.41.1 ENDWHILE Parameters

There are no ENDWHILE parameters.

7.41.2 ENDWHILE Examples

Refer to the [7.31.2 DOWHILE Examples](#) for examples of using this command.

7.42 EXCHANGE

The EXCHANGE command allows information to be exchanged between functions via an internal LANSAs queue called the "exchange list".

The "exchange list" is like a notice board that can be used to exchange information between functions. It is not the same as the parameter list concept that has been traditionally used to exchange information between programs.

When a function executes an EXCHANGE command (or uses an EXCHANGE parameter) it is in effect "tacking" information onto the notice board. This information is left on the notice board until another function is invoked.

When another function is invoked the first thing it does is to look at the entries on the notice board. If any of the entries "apply" to the function they are copied from the notice board into the function.

The notice board is then cleared. It is important to remember this step is always performed.

Note that an EXCHANGE command in a Component is only valid for passing information to and from a Function. For example, using the CALL or SUBMIT command. It is not valid to receive information into a Component this way. Unpredictable results will occur. (an example is that the instantiation of a Component does not clear the exchange list) The recommended way to pass information into and out of a Component is to use a method call or get a property or set a property.

Remember that there are many ways of invoking a function. These include being selected from the process main menu, by executing a TRANSFER command, a CALL command, a SUBMIT command or by using the PROMPT function key.

Also See

[7.42.1 EXCHANGE Parameters](#)

[7.42.2 EXCHANGE Comments / Warnings](#)

[7.42.3 EXCHANGE Accessing the Exchange List from RPG/COBOL](#)

[7.42.4 EXCHANGE Examples](#)

Optional

```
EXCHANGE ----- FIELDS ----- field name -----  
----->
```

| *expandable group expression* |
| |
----- 100 max -----

OPTION ----- *NOW -----|
*ALWAYS

7.42.1 EXCHANGE Parameters

FIELDS

OPTION

FIELDS

Specifies the name of the field(s) to be exchanged or the name of a group that defines the field(s) to be exchanged. For more details of how field and group names can be specified, refer to [Field Groups and Expandable Groups](#). Alternatively, an expandable group expression can be entered in this parameter. The contents of a BLOB/CLOB, not just the filename, will be moved when exchanging with a server or through Job Queues. This is because the file must be visible on the other end of the communications link.

OPTION

Specifies when the nominated fields are to be exchanged.

*NOW or N which is the default value, indicates that all nominated fields should be mapped into the exchange list now (ie: at this point in the program logic). This version of the EXCHANGE command is considered to be an executable command because it causes the fields to be placed into the exchange list at this point in the RDML program (and nowhere else).

*ALWAYS or A, the other allowable value, indicates that all nominated fields should be automatically mapped into the exchange list at several pre-defined points during the program's execution. These pre-defined points are:

- during any type of normal program termination
- before calling a process or function (CALL command)
- before invoking a PROMPT_KEY handling process or function
- before submitting a process or function (SUBMIT command)

The use of OPTION(*ALWAYS) creates a version of the EXCHANGE command that is considered to be a declarative command because it does not directly execute within the program. It declares that certain logic should be executed at other (pre-defined) points in the program. Usually this type of EXCHANGE command is coded at the beginning of the program, however it may be coded anywhere in the program.

7.42.2 EXCHANGE Comments / Warnings

- Information is exchanged between functions in an "exchange list". The format of the exchange list is something like this:

```
||N|T|L|D| V ||N|T|L|D| V | ..... |N|T|L|D| V ||
```

where:

N is the name of a field

T is the type of a field

L is the length of a field

D is the number of decimal positions

V is the variable length value of the field

- Whenever a function is invoked, a CALL to another function completes, or a PROMPT_KEY handling function completes after being prompted from an input capable field, the exchange list is searched. If a field is found in the exchange list with the same name as a field used in the function it is "mapped" into the function. After the search has been completed the exchange list is cleared, regardless of whether or not any fields were found in it and mapped into the function.
- It can be seen that the exchange of information between functions is by name, not by position as with normal program parameters.
- There are two exchange lists: RDML and RDMLX. In general, RDML Fields are put on the RDML exchange list and RDMLX fields are put on the RDMLX exchange list. The exception is when the current Function or Component is RDMLX. In this case, if the field is RDML but the RDML exchange list is full, the RDML field will be put on the RDMLX exchange list. For more information see [What Classifies a Field as RDMLX?](#)
- An RDML field can be exchanged from an RDMLX function or component to an RDML Function and back again provided the RDML exchange list is not full. If the exchange list is full, the RDML function will not exchange the value as the RDML field value will be on the RDMLX exchange list.
- The "mapping" procedure mentioned above will automatically convert field types, lengths and decimal positions if the definition of the field in the

exchange list is different to the definition of the field in the function. This does not mean that the conversion will always succeed. A 'best attempt' will be made to coerce the exchange value into the target field, and this may fail. The exchange list mechanism is designed for exchanging identical fields, and changing anything except the length is highly inadvisable.

When the type of the target field is Alpha or String and the length of the target field is shorter than the exchanged value, then the value is truncated to the maximum length of the target field.

When the type of the source field is Alpha or String, the Target Field is numeric and the length of the target field is shorter than the exchanged value, then the value is truncated to the left and right of the decimal point according to the length and decimals of the target field.

When the target and source fields are of any other combination and the length is shorter than the exchanged value, the behavior is undefined. This undefined behavior may also change in the future in an undefined way. You should assume that a result obtained with this undefined behavior will change in a future release.

- It is possible to exchange a field with the ASQN attribute. Note that it is expected that the field is defined identically (including the ASQN attribute) in both sets of code.
- When a PROMPT_KEY processing function is automatically invoked from a DISPLAY, REQUEST or POP_UP command (ie: when the prompt function key is used), the following events occur:
 - All fields nominated in EXCHANGE OPTION(*ALWAYS) commands are placed into the exchange list.
 - The field being prompted is placed into the exchange list.
 - If the field being prompted refers to another field for its definition, it is added to the exchange list again, but this time under the "refer" field name.
 - Special fields PROMPT\$FN and PROMPT\$RN are placed onto the exchange list. These fields contain the name of the field that is being prompted and the name of its associated reference field (if any) respectively.
 - As many of all the other fields used by the program as will fit into the space left in the exchange list are also added to the exchange list if the flag in the System Definition Data Area is set accordingly. If not, only the EXCHANGED fields will be placed on the exchange list. Refer to [The System Definition Data Areas](#) in the *LANSA for i User Guide*.
- The prompt key processing function is invoked.

- If the prompt key processing function completes normally

and

the field being prompted was input capable on the screen at the time it was prompted, fields placed onto the exchange list by the prompting function are mapped back into the calling program's storage.

- Finally the exchange list is always cleared of all entries.
- The EXCHANGE command or the EXCHANGE parameter on the TRANSFER command are used to add a new entry into the EXCHANGE list.
- The net length of the RDML field exchange list cannot exceed 2000 characters at any time or the EXCHANGE or TRANSFER command that is attempting to add information to the list will end abnormally.
- The length of the RDMLX exchange list is only limited by available memory.
- When working with the function that receives the EXCHANGE information remember that the exchange of information takes place before the first RDML command in the function is executed (ie: during the function initialization procedures).
- If the EXCHANGE command or parameter does not appear to be working correctly it is probably because the first RDML command in the function sets the fields that have just been mapped from the exchange list to *DEFAULT or *NULL. This causes the EXCHANGE values to be overwritten.
- When retrieving fields (GET) from the exchange list in RPG / CL etc. use the same field length and number of decimal positions. If different lengths are used unpredictable results may occur.
- Shorter field lengths on GET:
 - The decimal and / or integer parts may be truncated or trimmed.
 - Characters may be trimmed.
- Longer field lengths on GET:
 - Values will be padded with zeroes or blanks.

The introduction of the CALL PROCESS(*DIRECT) option has now made high volume calls possible. This was previously not recommended in LANSA. When implementing high volume calls to other functions, the size of the exchange list should be considered. In these circumstances, a large number of fields on the exchange list may cause a performance overhead. Investigate the option of passing data structures (CALL PASS_DS(#dddddd)) between functions

instead of using the exchange option.

7.42.3 EXCHANGE Accessing the Exchange List from RPG/COBOL

A flag field in the system definition data area DC@A01 must be set to indicate that the exchange capability is required. Refer to [The System Definition Data Area DC@A01](#) in the *LANSA for i User Guide*.

Normally the exchange list is only used to communicate information between LANSAs processes and functions, however, a facility does exist to allow RPG/COBOL/etc programs that call LANSAs applications or are called by LANSAs applications to access the exchange list.

This facility is actually a program called M@EXCHL that is shipped with LANSAs. By placing calls to M@EXCHL, RPG/COBOL/etc programs are able to place things onto the exchange list and get things off the exchange list, just like LANSAs processes and functions do.

You cannot use the M@EXCHL facility with RDMLX functions because X_RUN is not aware of external exchange lists. You can only CALL a LANSAs RDML function and not RDMLX if the exchange list is needed. That is,

- RPG calling LANSAs - this must be an RDML function
- LANSAs calling RPG - this could be either an RDML or RDMLX function. To do this, you need to create a simple RDML function for the MC@EXCHL values and then they can be exchanged to the RDMLX function. That is, call this RDMLX function from RDML to exchange the values.

Following are two different scenarios that demonstrate how to use M@EXCHL.

Scenario 1 - RPG/COBOL/etc calling a LANSAs application

The RPG/COBOL/etc program would have logic like this:

1. CALL M@EXCHL to "CLR" any rubbish left on the list.
2. CALL M@EXCHL to "PUT" information into the list.
3. CALL LANSAs to run the LANSAs function. This must be an RDML function, not an RDMLX function (called via LANSAs X_RUN).
4. CALL M@EXCHL to "GET" information from the list.

The actual logic flow of these operations works like this:

1. M@EXCHL clears any rubbish on the user list.
2. M@EXCHL places requested details into the user list.

3. As LANSAs is invoked, it appends the user list to the current exchange list and clears the user list. The function then runs under the normal exchange list rules. Just before control is returned to the RPG/COBOL/etc application, the exchange list is mapped into the user list and the exchange list is cleared.
4. M@EXCHL searches the user list and returns information.

Scenario 2 - A LANSAs application calling RPG/COBOL/etc

1. A LANSAs function places a call to an RPG/COBOL/etc program via the RDML CALL command. Since this program wishes to access the exchange list, the PGM_EXCH(*YES) parameter is used. This may be an RDML or RDMLX function, but only RDML fields can be exchanged. Refer to the [CALL](#) command for further information.
2. The RPG/COBOL/etc program receives control.
3. It calls M@EXCHL to "GET" details from the list.
4. It calls M@EXCHL to "CLR" the list.
5. It does whatever processing is required.
6. Then it calls M@EXCHL to "PUT" details back onto the list.
7. Control returns to the LANSAs function, and details from list appear and are processed just like an exchange via the exchange list with another LANSAs function.

The actual logic flow of these operations works like this:

1. Any OPTION(*ALWAYS) fields are added to the exchange list, and then the exchange list is mapped into the user list, then the exchange list is cleared, and the CALL to the RPG/COBOL/etc application is placed.
2. The RPG/COBOL/etc program receives control.
3. M@EXCHL searches the user list and returns information.
4. M@EXCHL clears the user list.
5. As above.
6. M@EXCHL places requested details into the user list.
7. Control returns to the LANSAs function, which replaces the exchange list with the user list, and clears the user list. Next the exchange list is searched for matches with fields in the function and, where required, mapping occurs. Finally, the exchange list is cleared.

Parameters required by M@EXCHL

M@EXCHL has 1 parameter that must be specified on a CALL, and up to 10 "pairs" of parameters that are used to define and contain information that is to be placed on or received from the exchange list. This means a minimum of 1 and a maximum of 21 parameters can be passed to M@EXCHL on any call.

In detail these parameters look like this:

Parm Number Type Min Len Max Len Comments

01	Alpha 3	3	PUT = Put data onto list GET = Get data from list CLR = Clear list
02,04, 06,08, 10,12, 14,16, 18,20	Alpha 15	15	Format : nnnnnnnnnntllld where: nnnnnnnnnn is the field name t is the field type (A/P/S) lll is the field length or the total digits d is the number of decimals
03,05, 07,09, 11,13, 15,17, 19,21	Any 1	256	This is the field's value.

Examples of calling M@EXCHL

Example 1: From a CL (control language) program place the value of field COMPNO (company number) and ACCTYP (account type) onto the exchange list. Run a LANSAL function and get back a field called TOTAMOUNT from the exchange list.

```
DCL &COMPNO *DEC (3 0)
DCL &ACCTYP *CHAR 5
DCL &TOTAMOUNT *DEC (13 2)
```

```
CALL M@EXCHL ('CLR')
```

```
CALL M@EXCHL ('PUT' 'COMPNO P0030' &COMPNO ACCTYP A005
```

```
LANSA RUN ... etc .....
```

```
CALL M@EXCHL ('GET' 'TOTAMOUNT P0132' &TOTAMOUNT)
```

Example 2: Do the same thing from RPG.

```
E          ATR   1  3 15
C*
C* CLEAR THE LIST
C*
C          CALL 'M@EXCHL'
C          PARM 'CLR'  A@EXCH 3
C*
C* PUT TO THE LIST
C*
C          CALL 'M@EXCHL'
C          PARM 'PUT'  A@EXCH
C          PARM      ATR,01
C          PARM      COMPNO 30
C          PARM      ATR,02
C          PARM      ACCTYP 5
C*
C* INVOKE THE FUNCTION
C*
C          CALL 'LANSA'
C          ----- ETC -----
C          ----- ETC -----
C          ----- ETC -----
C*
C* GET FROM THE LIST
C*
C* NOTE: The field is named "TOTAMOUNT" in the exchange
C* list, but is actually returned into a field
C* called TOTAMT.
C          CALL 'M@EXCHL'
C          PARM 'GET'  A@EXCH
C          PARM      ATR,03
```

```

C          PARM          TOTAMT 132
C*
C          MOVE '1'      *INLR
C          RETRN
**
COMPNO    P0030
ACCTYP    A0050
TOTAMOUNT P0132

```

Example 3: Write a CL program that is to receive the name of a file and library from the exchange list, copy the file to tape and place a return code back onto the exchange list.

```

COPYTAPE: PGM

DCL &FILE    *CHAR 10
DCL &LIBRARY *CHAR 10
DCL &RETCODE *CHAR 1
CALL M@EXCHL ('GET' 'FILE    A0100' &FILE  'LIBRARY  A0100' &I
CALL M@EXCHL ('CLR')

CHGVAR &RETCODE 'Y'
CPYTOTAP ... etc .....
MONMSG (CPF0000 MCH0000) EXEC(CHGVAR &RETCODE 'N')

CALL M@EXCHL ('PUT' 'RETCODE  A0010' &RETCODE)

ENDPGM

```

Example 4: Write the RDML code required to display a screen panel, input the file and library name, call the CL program in example 3 and act upon the return code.

```

DEFINE #FILE *CHAR 10
DEFINE #LIBRARY *CHAR 10
DEFINE #RETCODE *CHAR 1

REQUEST FIELDS(#FILE #LIBRARY)
EXCHANGE FIELDS(#FILE #LIBRARY)
CALL    PGM(COPYTAPE) PGM_EXCH(*YES)

```

```
IF      (#RETCODE *NE Y)  
MESSAGE MSGTXT('Copy to tape failed')  
ENDIF
```

7.42.4 EXCHANGE Examples

The following example applies to the EXCHANGE command in general, **not** to the program M@EXCHL:

Consider 2 functions being used that are named FUNC1 and FUNC2 and have RDML programs that look like this:

Function: FUNC1

```
REQUEST  (#FIELD01 #FIELD02)
INSERT   FIELDS(#FIELD01 #FIELD02) TO_FILE(FILE01)
```

Function: FUNC2

```
REQUEST  (#FIELD01 #FIELD02 #FIELD03 #FIELD04)
INSERT   FIELDS(#FIELD01 #FIELD02 #FIELD03 #FIELD04) TO_FILE(I
```

If FUNC1 had the following command added to it (as the last command) and was recompiled:

```
EXCHANGE FIELDS(#FIELD01 #FIELD02)
```

then, if the user used function FUNC1, and then used function FUNC2, the fields #FIELD01 and #FIELD02 would come up on the FUNC2 REQUEST display "prefilled" with the values entered by the user in function FUNC1.

There is no need to re-compile FUNC2 to make this happen. Remember that the exchange list is searched whenever a function is invoked. As soon as fields #FIELD01 and #FIELD02 start to appear in the exchange list they will be "mapped" into function FUNC2.

If the exchange command described above was also added to FUNC2 and FUNC2 was re-compiled then the following will happen.

Whenever FUNC2 is invoked after FUNC1 then the #FIELD01 and #FIELD02 values on the FUNC2 REQUEST display will be pre-filled with the values entered by the user in FUNC1.

Whenever FUNC1 is invoked after FUNC2 then the #FIELD01 and #FIELD02 values on the FUNC1 REQUEST display will be pre-filled with the values entered by the user in FUNC2.

If FUNC1 is invoked repeatedly from the process menu then the #FIELD01 and #FIELD02 values will be pre-filled with the values entered during the previous

invocation.

In other words FUNC1 is exchanging information with itself.

The first time that FUNC1 is invoked (assuming that FUNC2 has not been used already) the exchange list will be empty. In this case #FIELD01 and #FIELD02 will be pre-filled with the default values specified for them in the data dictionary or DEFINE command.

7.43 EXEC_CPF

This command has been deprecated. Please use [7.44 EXEC_OS400](#) instead.

7.44 EXEC_OS400

The EXEC_OS400 command is used to execute an IBM i operating system command from within an RDML or RDMLX program. Optionally fields from the function can be specified within the command and will be substituted with their values at execution time.

Portability Considerations This command is only supported in RDMLX programs for compatibility with existing RDML code. As such, it only supports the substitution of RDML field values at execution time.

Use the SYSTEM_COMMAND Built-In Function as a better alternative for RDMLX programs.

Also See

[7.44.1 EXEC_OS400 Parameters](#)

[7.44.2 EXEC_OS400 Comments / Warnings](#)

[7.44.3 EXEC_OS400 Example](#)

Required

```
EXEC_OS400 ----- COMMAND -----  
- 'AS/400 command' ----->
```

Optional

```
>-- IF_ERROR ----- *ABORT -----|  
      *NEXT  
      *RETURN  
      label
```

7.44.1 EXEC_OS400 Parameters

COMMAND

IF_ERROR

COMMAND

Specifies the IBM i (EXEC_OS400) command string that is to be executed from within the function. The command specified must be eligible for execution via the IBM supplied command execution programs QCMD EXC.

IF_ERROR

Specifies what action is to be taken if an error occurs when the command is executed.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the error.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of these values are used you must nominate a valid command label to which control should be passed.

7.44.2 EXEC_OS400 Comments / Warnings

The command string is executed via IBM supplied program QCMDEXC. As such it must be a valid command in IBM i format. This can only be executed on an IBM i.

Where an operating system command specified for execution via the EXEC_OS400 command fails to execute, place the function in LANSAs debug mode and attempt the operation again. Refer to [LANSA Debugging Mode](#) in the *LANSA for i User Guide* for details.

When a **function is in debug mode** the routine that actually handles the execution of the command will stop processing just before executing the command. The command that is about to be executed will be displayed in full. This includes the substitution values of any LANSAs fields that were imbedded into the command.

Fields from the function specified in the command string must be in **uppercase**. The process by which fields in the command string are substituted is quite complex. The following example shows most of the substitution methods used. If the command to be executed is:

```
EXEC_OS400 COMMAND('CALL PGM001 (#BATCH #TRANS)')
```

where #BATCH is a numeric field (7 0) and #TRANS is an alphanumeric field of length 10, this table indicates the actual command executed for various values of #BATCH and #TRANS:

Value in #Batch	Value in #Trans	Actual Command String Executed / Comments
42	XXXXXXXXXX	CALL PGM001 (X'0000042F' XXXXXXXXXXXX)
108	XXX XXX	CALL PGM001 (X'0000108F' 'XXX XXX ') because #TRANS contains imbedded blanks.
9999	1234567890	CALL PGM001 (X'0009999F' '1234567890') because #TRANS contains all numeric characters.

Note that the command specified must be enclosed in quotes. This means that if

the command itself must contain quotes then 2 will have to be used instead of one. For details of quotes and quoted strings, read the section on handling [Quotes and Quoted Strings](#).

OS400 Authority Considerations

Commands executed via EXEC_OS400 adopt the authority of the LANSAsystem owner user profile, and the user profile of any other entries in the call stack with USRPRF(*OWNER) as long as the "chain" is not broken by an entry in the call stack with USEADPAUT(*NO).

If this does not suit your site security policy, issue the commands:

```
CHGPGM PGM(M@CPEXEC) USRPRF(<your value>)  
USEADPAUT(<your value>)
```

```
CHGPGM PGM(M@OSEXEC) USRPRF(<your value>)  
USEADPAUT(<your value>)
```

If either of these commands fail, contact your LANSAsystem product vendor. You should also examine these values after any form of upgrade to your LANSAsystem.

7.44.3 EXEC_OS400 Example

Example 1: Write an RDML program that will execute any IBM i operating system command that can be executed via program QCMDEXC, and that will not abort if the command is invalid or contains errors:

```
DEFINE FIELD(#CMD) TYPE(*CHAR) LENGTH(200)
L1: REQUEST FIELDS(#CMD)
EXEC_OS400 COMMAND(#CMD) IF_ERROR(L1)
GOTO LABEL(L1)
```

7.45 EXECUTE

The EXECUTE command is used to execute a subroutine defined within a function and optionally pass parameters to it.

When the subroutine has completed execution control is passed to the command following the EXECUTE command.

Also See

[7.45.1 EXECUTE Parameters](#)

[7.45.2 EXECUTE Examples](#)

[7.38 ENDROUTINE](#)

[7.92 SUBROUTINE](#)

[EVTROUTINE](#)

[MTHROUTINE](#)

[PTYROUTINE](#)

Required

EXECUTE ----- *SUBROUTINE* --- *subroutine name* -----
----->

Optional

>-- *WITH_PARMS* --- *list of parameters* -----|
| *expandable group expression* |
----- *50 max* -----

7.45.1 EXECUTE Parameters

SUBROUTINE

WITH_PARMS

SUBROUTINE

Specifies the name of the subroutine that is to be executed. The subroutine named in this parameter must be defined elsewhere in the function with a SUBROUTINE command.

WITH_PARMS

Optionally defines a list of parameters that are to be passed to the subroutine. An expandable group expression is allowed in this parameter.

When executing a subroutine, the parameters specified in the WITH_PARMS parameter of the EXECUTE command must exactly match in number and type the parameters defined in the PARMs parameter of the associated SUBROUTINE command.

7.45.2 EXECUTE Examples

Refer to [7.92.3 SUBROUTINE Examples - Part 1](#) command for examples using this command.

7.46 EXIT

The EXIT command is used to cause an executing RDML program to end and an immediate exit from LANSAs to be performed. Note that the exit is from the entire LANSAs system, not just from the current function / process.

Using the EXIT command is functionally identical to using the EXIT function key.

Optionally a message may be issued which will be routed back onto the program message queue of the program that initially invoked LANSAs.

Also See

[7.46.1 EXIT Parameters](#)

[7.46.2 EXIT Examples](#)

Optional

```
EXIT ----- MSGTXT -----*NONE -----
-->
      'message text'

>-- MSGID ----- *NONE ----->
      message identifier

>-- MSGF ----- *NONE ----->
      message file . library name

>-- MSGDTA ----- substitution variables -----|
      | expandable group expression |
      ----- 20 max -----
```

7.46.1 EXIT Parameters

MSGDTA

MSGF

MSGID

MSGTXT

MSGTXT

Allows up to 80 characters of message text to be specified. This text will be routed back onto the program message queue of the program that initially invoked LANSAs. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID/MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be issued when the function ends and the exit from LANSAs is performed. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the

field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this:

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA(''BOLTS  '' #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.46.2 EXIT Examples

Example 1: Exit from a function with a text message:

```
EXIT MSGTXT('LANSA ended at user request')
```

Example 2: Exit from a function using a message identifier and message file:

```
EXIT MSGID(USR0046) MSGF(QUSRMSG)
```

```
EXIT MSGID(USR0167) MSGF(QUSRMSG.*LIBL)
```

```
EXIT MSGID(USR8046) MSGF(QUSRMSG.USERLIB01)
```

7.47 FETCH

The FETCH command is used to fetch fields from the first record in a file that matches a nominated key, condition or relative record number.

Portability Considerations Refer to parameters: [FROM_FILE](#) and [LOCK](#)

Also See

[7.47.1 FETCH Parameters](#)

[7.47.2 FETCH Examples](#)

Required

FETCH ----- *FIELDS* ----- *field name field attributes* -
-->

```
|          |          ||  
|          --- 7 max ----- |  
|*ALL          |  
|*ALL-REAL          |  
|*ALL-VIRT          |  
|*INCLUDING          |  
|*EXCLUDING          |  
| expandable group          |  
|----- 1000 max for RDMLX-----|  
|----- 100 max for RDML -----|
```

>-- *FROM_FILE* ---- *file name* . *FIRST -----

>

library name

Optional

>-- *WHERE* ----- '*condition*' ----->

>-- *WITH_KEY* ---- *key field values* ----->
expandable group expression

>-- *IO_STATUS* ---- **STATUS* ----->
field name

```
>-- IO_ERROR ----- *ABORT -----
>
    *NEXT
    *RETURN
    label

>-- VAL_ERROR ---- *LASTDIS -----
>
    *NEXT
    *RETURN
    label

->-- NOT_FOUND ---- *NEXT -----
->
    *RETURN
    label

>-- ISSUE_MSG ---- *NO ----->
    *YES

>-- LOCK ----- *NO ----->
    *YES

>-- WITH_RRN ----- *NONE -----
>

>-- RETURN_RRN --- *NONE -----
->

>-- KEEP_LAST ---- *NONE -----|
    1 - 9999
```

7.47.1 FETCH Parameters

FIELDS

FROM_FILE

IO_ERROR

IO_STATUS

ISSUE_MSG

KEEP_LAST

LOCK

NOT_FOUND

RETURN_RRN

VAL_ERROR

WHERE

WITH_KEY

WITH_RRN

FIELDS

Specifies either the field(s) that are to be fetched from the file or the name of a group that specifies the field(s) to be fetched. Alternatively, an expandable group expression can be entered in this parameter. For more details, refer to [Expandable Groups](#).

The following special values can be used:

- *ALL specifies that all fields from the currently active file be fetched.
- *ALL_REAL specifies that all real fields from the currently active file be fetched.
- *ALL_VIRT specifies that all virtual fields from the currently active file be fetched.
- *EXCLUDING specifies that fields following this special value must be excluded from the field list.
- *INCLUDING specifies that fields following this special value must be included in the field list. This special value is only required after an *EXCLUDING entry has caused the field list to be in exclusion mode.

Note: When all fields are fetched from a logical file maintained by OTHER, all the fields from the based-on physical file are included in

the field list.

It is strongly recommended that the special values *ALL, *ALL_REAL or *ALL_VIRT be used sparingly and only when required. Fetching fields which are not needed causes the function to retrieve and map fields unnecessarily, invalidates cross-reference details (shows fields which are not used in the function) and increases the Crude Entity Complexity Rating of the function pointlessly.

Note that when BLOB or CLOB data is retrieved, it is either *SQLNULL or a filename. If a filename, the data from the database file has been copied into the file.

Warning: It is time-consuming to retrieve BLOB or CLOB fields from a file.

Recommended Database Design When Using BLOB and CLOB Fields

The recommended design when using BLOB and CLOB fields is to put them in a separate file from the rest of the fields using the same key as the main file. This forces programmers to do separate IOs to access the BLOB and CLOB data, thus reducing impact on database performance from indiscriminate use of this data. It is also the most portable design ensuring that the non-BLOB and non-CLOB data can be quickly accessed at all times.

FROM_FILE

Refer to [Specifying File Names in I/O commands](#).

WHERE

Refer to [Specifying Conditions and Expressions](#) and [Specifying WHERE Parameter in I/O Commands](#).

After a fetch utilizing a where condition that results in a record not found, the contents of the fields are unpredictable.

WITH_KEY

Refer to [Specifying File Key Lists in I/O Commands](#).

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

For details of I/O operation return code values, refer to [I/O Return Codes](#).

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

VAL_ERROR

Specifies the action to be taken if a validation error was detected by the command.

A validation error occurs when information that is to be added, updated or deleted from the file does not pass the FILE or DICTIONARY level validation checks associated with fields in the file.

If the default value *LASTDIS is used control will be passed back to the last display screen used. The field(s) that failed the associated validation checks will be displayed in reverse image and the cursor positioned to the first field in error on the screen.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).

When using *LASTDIS the "Last Display" must be at the same level as the database command (INSERT, UPDATE, DELETE, FETCH and SELECT). If they are at different levels e.g. the database command is specified in a SUBROUTINE, but the "Last Display" is a caller routine or the mainline, the function will abort with the appropriate error message(s).

The same does NOT apply to the use of event routines and method routines in Visual LANSA. In these cases, control will be returned to the calling routine. The fields will display in error with messages returned to the first status bar encountered in the parent chain of forms, or if none exist, the first form with a status bar encountered in the execution stack (for example, a reusable part that inherits from PRIM_OBJT).

NOT_FOUND

Specifies what is to happen if no record is found in the file that has a key matching the key nominated in the WITH_KEY parameter.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

ISSUE_MSG

Specifies whether a "not found" message is to be automatically issued or not.

The default value is *NO which indicates that no message should be issued.

The only other allowable value is *YES which indicates that a message should be automatically issued. The message will appear on line 22/24 of the next screen format presented to the user or on the job log of a batch job.

LOCK

Specifies whether or not the record should be locked when it is read.

*NO, which is the default value, indicates that the record should not be locked.

*YES indicates the record should be locked. It is the responsibility of the user to ensure that the record is released at some future time.

NOTE: LOCK(*YES) performs a record level lock. It may exhibit intra and inter operating system behavioural variations (e.g. commitment control locking multiple records; default wait times). User's are advised to investigate the development of proper and complete "user object" locking protocol by using the LOCK_OBJECT Built-In Function.

Portability Not supported and should not be used in portable applications.
Considerations A build warning will be generated when used in Visual LANSA.

WITH_RRN

Specifies the name of a field that contains the relative record number (for relative record file processing) of the record which is to be fetched. The WITH_RRN parameter cannot be used if the WITH_KEY or WHERE parameters are used.

Any field nominated in this parameter must be defined within the function or the LANSA data dictionary and must be numeric.

Note: Using the WITH_RRN parameter to FETCH, DELETE or UPDATE records is faster than any other form of **database** access.

The actual database file being accessed can not be a logical file when using the WITH_RRN parameter.

For information, refer also to [Load Other File](#) in the *Visual LANSA Developers Guide*.

RETURN_RRN

Specifies the name of a field in which the relative record number of the record just fetched should be returned.

Any field nominated in this parameter must be defined within the function or the LANSA data dictionary and must be numeric.

For further information refer also to [Load Other File](#) in the *Visual LANSA Developers Guide*.

KEEP_LAST

Specifies that details of the last "n" FETCH operations be kept within the compiled RDML program. Whenever a FETCH command is executed the details of the last "n" FETCH operations are searched first. If the required details are found within the program, no database I/O operation actually occurs. This can dramatically improve the performance of RDML programs.

*NONE, which is the default value, indicates that no details of previous FETCH operations are to be kept. Every FETCH performed will result in I/O to the associated database file.

An integer in the range 1 to 9999 may be specified. This indicates the number of previous FETCH operations for which details should be kept. The number specified reflects the maximum number of different FETCH operations that could be reasonably expected.

For instance, if the FETCH is made to a company file (by key company number) to get a company name, and only 15 companies exist, a value of 15 would be correct. Specifying 500 will waste storage in the executing RDML program and may actually degrade its performance.

Note that the number specified does not limit the number of different FETCH operations allowed. If no space is available within the RDML program for a FETCH's details to be stored, the oldest FETCH details are overwritten.

When the KEEP_LAST parameter is specified, the WITH_KEY parameter must be specified and the WHERE parameter must not be specified.

Note also that since previous FETCH details are stored within the RDML program, it is possible for the actual database file details to be changed without the change being reflected in the RDML program.

7.47.2 FETCH Examples

Example 1: Fetch fields #NAME, #ADDL1 and #POSTCD from the record in file CUSTMST that has key #CUSNUM:

```
FETCH    FIELDS(#NAME #ADDL1 #POSTCD) FROM_FILE(CUSMST)
```

or identically:

```
GROUP_BY  NAME(#CUSTOMER) FIELDS(#NAME #ADDL1 #POSTCD)
FETCH    FIELDS(#CUSTOMER) FROM_FILE(CUSMST) WITH_KEY(#
```

Example 2: Fetch a tax rate (#TAXRATE) from a table of valid tax codes. The first key to the table indicates the tax type which in this case is always "income tax" and the second is the tax code (#TAXCDE):

```
FETCH    FIELDS(#TAXRATE) FROM_FILE(TAXTAB) WITH_KEY('INCC
```

Example 3: Fetch the product number (#PRODUCT) of the first order in an order line files where the #QUANTITY field is greater than 10:

```
FETCH    FIELDS(#PRODUCT) FROM_FILE(ORDLIN) WHERE(#QUANT
```

Example 4: Read and print details of all general ledger transactions from a file called GLTRANS. Associated with each transaction is a company number (#COMPNO). Fetch the actual company name (#COMPNAME) from file COMPANY and include this on the report:

```
GROUP_BY  NAME(#REPORTLIN) FIELDS(#TRANSNUM #TRANSTYP

SELECT    FIELDS(#REPORTLIN) FROM_FILE(GLTRANS)
FETCH    FIELDS(#COMPNAME) FROM_FILE(COMPANY) WITH_KEY(
UPRINT    FIELDS(#REPORTLIN)
ENDSELECT

ENDPRINT
```

Note that if there were 10,000 transactions in GLTRANS, and 15 (or less) different companies, this program would perform at most 10,015 database I/O operations. If the KEEP_LAST parameter on the FETCH command was omitted it would perform 20,000 I/O operations, which would probably double its

execution time.

Example 5: Fetch all fields from the currently active version of file CUSMST with key #CUSNUM:

```
FETCH    FIELDS(*ALL) FROM_FILE(CUSMST) WITH_KEY(#CUSNU.
```

Example 6: Fetch all real fields from the currently active version of file CUSMST but exclude address information:

```
GROUP_BY  NAME(#XG_ADDR) FIELDS(#ADDL1 #ADDL2 #ADDL3 #  
FETCH    FIELDS(*ALL_REAL *EXCLUDING #XG_ADDR) FROM_FIL
```

7.48 FILECHECK

The FILECHECK command is used to check a field against an entry in a file.

Portability Considerations Refer to parameter [USING_FILE](#) .

Also See

[7.48.1 FILECHECK Parameters](#)

[7.48.2 FILECHECK Comments / Warnings](#)

[7.48.3 FILECHECK Examples](#)

[7.4 BEGINCHECK](#)

[7.7 CALLCHECK](#)

[7.14 CONDCHECK](#)

[7.16 DATECHECK](#)

[7.35 ENDCHECK](#)

[7.78 RANGECHECK](#)

[7.99 VALUECHECK](#)

Required

```
FILECHECK ---- FIELD ----- field name -----  
--->
```

```
>-- USING_FILE --- file name . *FIRST -----  
>
```

library name

Optional

```
>-- USING_KEY ---- *FIELD -----  
>
```

key field values
expandable group expression

```
>-- FOUND ----- *NEXT ----->  
*ERROR
```

**ACCEPT*

--> >-- *NOT_FOUND* ---- **ERROR* -----

**NEXT*

**ACCEPT*

>-- *MSGTXT* ----- **NONE* ----->
message text

>-- *MSGID* ----- *DCU0003* ----->
message identifier

> >-- *MSGF* ----- *DC@M01 . *LIBL* -----
message file . library name

>-- *MSGDTA* ----- *substitution variables* -----|
|expandable group expression |
----- 20 max -----

7.48.1 FILECHECK Parameters

FIELD

FOUND

MSGDTA

MSGF

MSGID

MSGTXT

NOT_FOUND

USING_FILE

USING_KEY

FIELD

Specifies the name of the field which is to be associated with the check.

USING_FILE

Specifies the name of the file that is to be looked up by this check. Refer to [Specifying File Names in I/O commands](#).

USING_KEY

Specifies the key that is to be used to look up the file specified in the USING_FILE parameter.

*FIELD, which is the default value indicates that the field nominated in the FIELD parameter should be used as the key to look up the file.

To specify other key field(s) refer, for more details, to [Specifying File Key Lists in I/O Commands](#).

FOUND

Specifies the action to be taken if a record is found in the nominated file with the nominated key.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be

positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

NOT_FOUND

Specifies the action to be taken if no record can be found in the nominated file with the key value specified.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file

name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this:

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA(''BOLTS  '' #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.48.2 FILECHECK Comments / Warnings

- The FILECHECK command only checks for the presence of a matching key. The record from the file is **not** read into the program. Refer to the FETCH command if it is necessary to read a record into the program.
- FILECHECK commands must be within a BEGINCHECK / ENDCHECK validation block. Refer to these commands for further details.

7.48.3 FILECHECK Examples

[Structuring Functions for Inline Validation](#)

[Structuring Functions to Use a Validation Subroutine](#)

[Using the FILECHECK Command for Inline Validation](#)

[Using the FILECHECK Command for Validation with a Subroutine](#)

Structuring Functions for Inline Validation

Typically functions using validation commands (eg: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for inline validation like this:

```
BEGIN_LOOP
REQUEST  << INPUT >>
BEGINCHECK
*        << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK
*        << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is passed back to the REQUEST command. This happens because of the default IF_ERROR(*LASTDIS) parameter on the ENDCHECK command.

Structuring Functions to Use a Validation Subroutine

Typically functions using validation commands (eg: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for subroutine validation like this:

```
DEFINE   FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')

BEGIN_LOOP
DUNTIL  COND(*NOERRORS)
REQUEST << INPUT >>
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
```

```
*      << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE   FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
*      << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is returned to the main function loop with #ERRORCNT > 0.

Using the FILECHECK Command for Inline Validation

This example demonstrates how to use the FILECHECK command within the main program block to check an employee number against entries in a personnel file.

```
DEF_LIST  NAME(#EMPBROWSE) FIELDS(#EMPNO #GIVENAME #SU

BEGIN_LOOP
REQUEST   FIELDS(#EMPNO #GIVENAME #SURNAME) BROWSELIST

BEGINCHECK
FILECHECK FIELD(#EMPNO) USING_FILE(PSLMST) FOUND(*ERROR
ENDCHECK

ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

If the value of #EMPNO is found in the file PSLMST the message defined with the FILECHECK command is issued and program control returns to the last screen displayed. In this case the last screen displayed is the REQUEST screen.

Using the FILECHECK Command for Validation with a Subroutine

This example demonstrates how to use the FILECHECK command inside a subroutine to check an employee number against entries in a personnel file.

After the user enters the requested details the VALIDATE subroutine is called. It

checks that the value of #EMPNO is not already present in the PSLMST file. If this is true the message defined in the FILECHECK command is given and the DOUNTIL loop executes again. When a #EMPNO value is entered that is not found in the file the DOUNTIL loop ends and processing of the verified input is done.

```
DEFINE FIELD(#ERRORCNT) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #GIVENAME #SU
```

```
BEGIN_LOOP
DUNTIL COND(*NOERRORS)
REQUEST FIELDS(#EMPNO #GIVENAME #SURNAME) BROWSELIST
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
```

```
BEGINCHECK KEEP_COUNT(#ERRORCNT)
FILECHECK FIELD(#EMPNO) USING_FILE(PSLMST) FOUND(*ERROR)
ENDCHECK IF_ERROR(*NEXT)
```

```
ENDROUTINE
```

7.49 FUNCTION

The FUNCTION command is used to specify certain compilation options that affect the way an RDML program is generated and thus the way it behaves when it is actually being executed.

By specifying certain values with this command, the behavior of an RDML function can often be altered to produce better performance characteristics in a specific operating environment.

Portability Considerations

Note Visual LANSA considerations in this command's parameters.

Also See

[7.49.1 FUNCTION Parameters](#)

[7.49.2 FUNCTION Examples](#)

Optional

*FUNCTION ----- OPTIONS ----- function control option --
----->*

**NOMESSAGES
*DEFERWRITE
*HEAVYUSAGE
*LIGHTUSAGE
*DBOPTIMISE
*DBOPTIMIZE
*DBOPTIMISE_BATCH
*DBOPTIMIZE_BATCH
*PGMCOMMIT
*NOPGMCOMMIT
*NOIGCCNV
*NO_RLTB_MIRROR
*DIRECT
*CLOSE_DISPLAY
*MLOPTIMISE
*MLOPTIMIZE
*ALP_SYSTEM_VARIABLE
*NUM_SYSTEM_VARIABLE
ALP_FIELD_VALIDATE

**NUM_FIELD_VALIDATE*

**MINI_SCREEN*

**OS400_EXT_PRINT*

**BUILTIN*

| |
--- 9 max ---

>-- *RCV_DS* ----- *data structure names* ----->

| |
----- 20 max -----

>-- *RCV_LIST* ----- *working list names* ----->

| |
----- 20 max -----

>-- *TRIGGER* ----- **NONE* ----- *name* -----

|

**FIELD*

**FILE*

type

7.49.1 FUNCTION Parameters

OPTIONS

RCV_DS

RCV_LIST

TRIGGER

OPTIONS

Allows up to 9 different options to be specified. Values that may be specified in this parameter include:

*NOMESSAGES

Specifies that the program will never be required to route messages in from its caller, nor route messages back to its caller (unless it fails). By using this option the entry and exit resources used by a compiled RDML function can be reduced. When this option is used, outstanding developer messages are **not** checked for. This can benefit performance of heavily used / called functions in a production environment where developer services is on.

*DEFERWRITE

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA code.

Specifies that any IBM i display file created to service DISPLAY, REQUEST or POP_UP commands within this program should always use the DFRWRT(*YES) parameter. By using this option the time spent by the program waiting for device responses can be reduced.

Any program that uses a POP_UP command, or communicates to remotely attached devices, **should use** this option.

*HEAVYUSAGE and *LIGHTUSAGE

Specify that the compiled RDML function should use the HEAVY usage option or the LIGHT usage option regardless of what option the associated process uses.

For details of the heavy and light usage options, refer to [Anticipated Usage](#).

Note that RDMLX functions will, by default, retain their state between invocations. If the state is not to be retained, then use components that are *DYNAMIC.

***DBOPTIMIZE or *DBOPTIMISE**

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA and RDMLX code.

Specifies that the RDML function should **not** use I/O modules to access database files, but rather OPTIMIZE database access by using direct I/O techniques.

Using this option can enhance application performance. However, many additional considerations and restrictions apply to using this option.

It is **strongly recommended** that you read [Using *DBOPTIMIZE / *DBOPTIMIZE_Batch](#) in the *LANSA for i User Guide* before attempting to use this option.

***DBOPTIMIZE_BATCH or *DBOPTIMISE_BATCH**

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA and RDMLX code.

Specifies that the RDML function should **not** use I/O modules to access database files, but rather OPTIMIZE database access by using direct I/O techniques best suited to batch processing involving large volumes of update or delete operations.

Using this option can enhance batch application performance. However, many additional considerations and restrictions apply to using this option.

It is **strongly recommended** that you read [Using *DBOPTIMIZE / *DBOPTIMIZE_Batch](#) in the *LANSA for i User Guide* before attempting to use this option.

***PGMCOMMIT**

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA and RDMLX code. A build warning will be generated.

Specifies that individual program level commitment control is required for **all** files opened for **any** type of update activity by this RDML function.

Using this option **overrides and supersedes** any individual database file's definition or RDML command with regard to commitment control status and/or autocommit options.

Additionally, the operating system's commitment control facility will be **started** and **ended** automatically by the RDML function. See [User Exit F@BGNCMT -](#)

[Start Commitment Control](#) and [User Exit F@ENDCMT - End Commitment Control](#) in the *LANSA for i User Guide* for details.

The issuing of COMMIT and ROLLBACK commands at the appropriate transaction boundary is the responsibility of the user.

This facility is primarily intended for batch processing.

It is **strongly recommended** that [Commitment Control](#) in the *LANSA for i User Guide* be read in full before attempting to use this option.

Using *PGMCOMMIT **implies** the use of *DBOPTIMIZE, regardless of whether or not the *DBOPTIMIZE option is actually specified.

***NOPGMCOMMIT**

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA and RDMLX code. A build warning will be generated.

Specifies that individual program level commitment control is NOT required for **all** files opened for **any** type of update activity by this RDML function.

Using this option **overrides and supersedes** any individual database file's definition or RDML command with regard to commitment control status and/or autocommit options.

Using *NOPGMCOMMIT **implies** the use of *DBOPTIMIZE, regardless of whether or not the *DBOPTIMIZE option is actually specified.

***NOIGCCNV**

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA code.

Specifies that the IGCCNV DDS keyword (for IGC conversion) should **not** be enabled for any display file created to support this function, regardless of the setting of the "IGCCNV required" flag in the definition of the current language.

Normally, any display file created for a function, under a language that has the "IGCCNV required" flag set, has the IGCCNV DDS keyword generated into it automatically.

Using this option suppresses the automatic enabling of the IGCCNV keyword in all display file DDS generated for this function.

***NO_RLTB_MIRROR**

Specifies that the automatic "mirroring" of field positions on screen panels and report layouts should **not** be enabled in this function, regardless of whether or

not the function is being compiled under a right-to-left language.

The automatic "mirroring" facility, and this parameter value, only apply to functions being compiled under right-to-left languages. This parameter is ignored for all other language groups.

***DIRECT**

Specifies that this function should be made eligible for potential direct calling from another function, or to directly service a prompt key request.

Note: All RDMLX Functions must use *DIRECT. This ensures that migrated IBM i RDML Functions are unique.

By using this option you are indicating that this function **may** be directly invoked by another caller function, or to directly service a prompt key request.

Whether or not this is a completely valid way to invoke this function is not important at this stage. This option just indicates that, should the need arise, the function may be directly called.

It is recommended that all functions include a FUNCTION command containing this option, and that any application template created before this option became available, should be modified to automatically generate a FUNCTION command using this option.

Refer to the CALL command for more details of how a direct mode call is made and the restrictions that exist when using this type of call operation.

***CLOSE_DISPLAY**

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA code.

This option indicates that even though the function may remain active as a HEAVY usage process, or a *HEAVYUSAGE function, its display file should be closed when it terminates, and re-opened when it is activated again.

This option is primarily intended for use in pop-up window prompt key functions that suffer from "restored" displays that are out of date, or that "flash" onto the screen.

If this option is used, ensure that all **browse** lists are specifically cleared (CLR_LIST command) at each entry or (re)entry to the function. This ensures that any counter fields are reset to zero to match the current number of entries in the list, which will be zero because the display file was closed on any previous termination.

***MLOPTIMIZE or *MLOPTIMISE**

**Portability
Considerations**

Will be ignored if used in Visual LANSA and RDMLX code.

Specifies that an RDML function using multilingual support (which is defined at the partition level) should be optimized for multilingual application support of five or less languages.

Using this option can enhance application performance where typically five or less languages are being supported.

When an RDML function is compiled a main program object results.

When the RDML function is in a multilingual partition an "extra" program object is also produced.

The main RDML compiled function declares a storage area to contains all "literal" values that may be subject to dynamic change by language.

When it is invoked it calls the extra program to initialize the storage area with the literal values that are correct for the current language.

This is an efficient approach when quite a large number of languages are involved because the main program only has to declare storage sufficient for one language set.

When the extra program is invoked it temporarily uses storage sufficient for all languages, copies the correct language details into the main programs storage area, then ends, freeing all the extra (and now unneeded) language storage areas.

This approach also has two disadvantages. Firstly, it means that the RDML function takes longer to compile because two compiled program objects must be produced. Secondly, it means that the main RDML function must place a call to the extra initializing program during function startup.

By using *MLOPTIMIZE (or *MLOPTIMISE) the existence of the extra initializing function can be suppressed. The storage required for all languages is declared in the main program and the storage area used for the current language is initialized directly by the main program.

Using *MLOPTIMIZE in a function means more storage requirements in the main program, but less resource usage during compiles and during function invocation.

Since more storage is used, it is recommended that *MLOPTIMIZE is only used in functions that are supporting 5 or less languages.

However, the value 5 is a recommendation only, and the option can be used in functions supporting more languages, at the discretion of the application

designer.

The following points about *MLOPTIMIZE should also be noted:

- You must be using OS/400 V2R1 (or higher).
- The application must be in a multilingual partition.

Use of *MLOPTIMIZE in any situation where these conditions are not all met does no harm. A warning message is issued and the *MLOPTIMIZE request is ignored.

***ALP_SYSTEM_VARIABLE**

Specifies that this function is to be a system variable evaluation function (for alphanumeric variables only). Refer to the [System Variable Evaluation Programs](#) in the *Visual LANSA Developer Guide* for more information. Option *DIRECT must also be used when this option is used.

***NUM_SYSTEM_VARIABLE**

Specifies that this function is to be a system variable evaluation function (for numeric system variables only). Refer to the [System Variable Evaluation Programs](#) in the *Visual LANSA Developer Guide* for more information and the design constraints on the use of functions for system variable evaluation. Option *DIRECT must also be used when this option is used.

***ALP_FIELD_VALIDATE**

Specifies that this function is to be a complex logic check function (for alphanumeric fields only).

Refer to the [Complex Logic Rule](#) in the *LANSA for i User Guide* for further information. Option *DIRECT must also be used when this option is used.

***NUM_FIELD_VALIDATE**

Specifies that this function is to be a complex logic check function (for numeric fields only).

Refer to the [Complex Logic Rule](#) in the *LANSA for i User Guide* for further information. Option *DIRECT must also be used when this option is used.

Technical notes for *ALP_FIELD_VALIDATE and

***NUM_FIELD_VALIDATE**

Complex logic validation functions can handle fields of different lengths and decimal precision but not of different types. The FUNCTION command option of *ALP_FIELD_VALIDATE will indicate that this is a function to validate an alphanumeric field. The FUNCTION command option of

*NUM_FIELD_VALIDATE will indicate that this is a function to validate a numeric field.

In order to access the field name, length and the field value within the validation function it is necessary to define the following fields in the data dictionary:

- VALFLD\$NM A(10) Name of Field being Validated
 - VALFLD\$LN P(7,0) Length of Field being Validated
 - VALFLD\$DP P(7,0) No. of decimals for Field being Validated
 - VALFLD\$AV A(256) Current Field Value (Alphanumeric Field and only the first 256 bytes of *Char/ *String Field).
- Note:** The *SQLNULL special value can't be evaluated to the *Char/*String Field Type in the CALLBACK Function because the VALFLD\$AV is *Alpha type.

-
- VALFLD\$NV P(30,9) Current Field Value (Numeric Field)
- VALFLD\$RT A(1) Return code

Note that this implementation effectively prohibits validating numeric fields that have more than 21 significant digits.

If option *ALP_FIELD_VALIDATE or *NUM_FIELD_VALIDATE are used within the function, return the calculated return code in field VALFLD\$RT. This should be returned by the validation function as '1' (good return) or '0' (bad return).

If either *ALP_FIELD_VALIDATE or *NUM_FIELD_VALIDATE have been entered as a function option the following **design** constraints (rather than technical constraints) exist to ensure the correct use of the facility:

- No DISPLAY, REQUEST or POP_UP command can be used within a complex logic validation function.
- No CALL can exist to another process/function within a complex logic validation function. However, a call to a 3GL program can exist.
- Complex logic validation functions cannot exist within an action bar process. This is not to say that they cannot be referenced from within an action bar, it just means that a complex logic function cannot be defined as part of a process that is of action bar type.
- Complex logic validation functions cannot have options of RCV_DS or RCV_LIST.
- The associated process must not have parameters.

- The exchange list may not be used. This restriction ensures insulated modularity in the validation check.
- Recursive implementations may be defined, but will fail to execute correctly. For instance a validation checker function invoked during an insert to file A could attempt to insert data into file B, possibly causing itself to be invoked in a recursive situation, and thus to fail.
- Use of options *DBOPTIMIZE and *NOMESSAGES are recommended for complex logic validation functions. The use of *HEAVYUSAGE may also be considered in heavily used validation functions.
- The use of option *MLOPTIMIZE is **strongly** recommended in all multilingual applications of this facility.

*MINI_SCREEN

Portability Do
Considerations not

use this option unless the application is

IBM i

based and it is using "miniature" or "palm top" devices that have a screen panel size smaller than the normal 24 line x 80 column devices.

Do

not

use this option in functions that contain normal full panel DISPLAY or REQUEST commands.

Do

not

use this option in functions that are GUI enabled. If used in Visual LANSA code it will be ignored. A build warning will be generated.

Specifies that POP_UP commands used within this function are to be used for "mini screen" display devices that may be attached to an IBM i system. When this option is used the following changes to the normal way that the POP_UP commands function are activated:

- No borders are presented.
- The window can be located left as far as row 1, column 1.

- The borderless window produces a representation of a "mini" full screen panel, rather than of a typical pop-up window.
- Browselists presented within a pop-up window where all field column headings have been overridden to blanks values will not be spaced/separated from the header area by the normal separation line.

This means that manually defined text can be effectively specified for "apparent" column headings. This facility is only used where **all** fields in the browselist have their column headings overridden to blanks. It also means that it is hard to insert a whole **new** line of text into the design via the screen painter because the normal separation line cannot be used as a target position to "push" the browselist down the screen panel. To insert a new line of text return to the RDML editor and define a "dummy" field into the header area at the end of the FIELDS list and then re-invoke the screen painter. The dummy field should be positioned so as to "push" the browselist further down the screen panel. Add/move the necessary text and/or field onto the new line created and then optionally delete the "dummy" field.

Where large scale use of this feature is being made it is **strongly recommended** that an Application Template be constructed for invocation by the "ET" (Execute Template) editor action. This template can be used to construct a "standard layout" for every "mini-screen" including one or more initial "dummy" fields to push the browselist portion onto the required starting line and leave sufficient space for the insertion of fields and/or text into the header area.

***OS400_EXT_PRINT**

Portability Not supported in Visual LANSA and RDMLX code. A build
Considerations warning will be generated if used.

specifies that LANSA should generate and use IBM i specific External Printer files for the RDML function. By itself there is no advantage in using this option, but it must be specified to utilize User Define Reporting Attributes within the RDML function.

The use of this Option makes the function **IBM i PLATFORM DEPENDENT**.

When *OS400_EXT_PRINT is specified there are certain restrictions:

- *DIRECT must also be specified.
- The SKIP command cannot be used.

- Only Report Number 1 may be used.

If any of these restrictions are broken then a fatal error will result when function Checking.

More information on *User Defined Reporting Attributes* and External Printer files, is supplied in [User Defined Reporting Attributes](#) in the *LANSA for i User Guide*.

***BUILTIN**

Specifies that this function is to be a Built-In Function.

Refer to [Create your Own Built-In Functions](#) in the *LANSA Application Design Guide* for more details. *DIRECT must also be used when this option is used.

***STRICT_NULL_ASSIGN**

Specifies that it is an error to assign an *SQLNULL into a non-nullable field.

The default is not to be strict and in simple terms treat an *SQLNULL value as *NULL when assigning into a non-nullable field. For a definition of the *NULL value for each of the field types, refer to [7.9.1 CHANGE Parameters](#).

Refer to [Assignment, Conditions, and Expressions with Fields allowing SQL Null](#) for full details on strict null assignment versus the default behaviour.

RCV_DS

Allows up to 20 different data structure names to be specified which can be received by the function. The following points should be noted when using this parameter:

Each data structure name should be the name of a physical file which has been defined to LANSAs.

FUNCTION OPTIONS(*DIRECT) must be specified when this parameter is used.

A function receiving data structure(s) is flagged as being not directly accessible from a process menu (or an action bar) during its compilation, unless the special *EXCHANGE option is used.

Such a function has to be called from another function that will pass the correct data structures (in the correct order) rather than being called directly from a process menu or action bar.

To be able to receive fields within the named physical file (ie: data structure), the fields must be referenced at some point within the function, otherwise they will not be received. This applies to the calling function also. Only real fields from the file can be passed, **not** virtual fields.

It is important to note that the order in which the data structures are specified on the PASS_DS parameter of the CALL command and the order in which they are specified on the RCV_DS on the FUNCTION command of the called function is significant - the data structures must appear in the same order in the called and calling functions, otherwise errors could occur.

Likewise, when the layout of a data structure is changed, all functions that reference it in a RCV_DS or PASS_DS parameter should be recompiled after the changed data structure has been made operational.

A **specialized** option called *EXCHANGE may be used as the first argument of the RCV_DS parameter (e.g.: RCV_DS(*EXCHANGE CUSMST PRODMST)). This indicates that the named data structures are to be passed and returned via an "exchange list" type of structure, rather than as actual parameters.

This facility is highly specialized and designed for use only in functions that exactly match the following criteria. **DO NOT USE** this option unless your function exactly matches these criteria:

- They are directly invoked from a menu or an action bar. This feature was provided to allow such functions to "exchange" a complete set of data structures amongst themselves, rather than having to use the EXCHANGE command to exchange many individual fields.
- They are not called by other functions. Why use this option when the data structure can be directly passed in by the caller in the normal manner?

The processing logic used in a function using RCV_DS(*EXCHANGE) is like this:

- As the function is entered, the "exchange area" is searched for data structures (by data structure name, not by order of specification). When a match is found the content is copied into the data structure. When a match is not found the content of the data structure remains unchanged.
- The "exchange area" is then cleared of all data structures.
- The function then proceeds to do its normal processing.
- When the function terminates normally (in any way) the contents of all the data structures nominated in the RCV_DS parameter are copied back into the "exchange area".

Some technical considerations when using this option are:

- Only functions that use RCV_DS(*EXCHANGE) will search, clear and alter the exchange area. Functions that do not use this option have no impact at all on the exchange area.

- The maximum length of the exchange area is 9999 bytes. Attempting to use a set of data structures with an aggregate length exceeding this limit will cause an application failure.
- The actual storage of the exchange area is performed by a shipped program called M@EXCHDS. To optimize performance under the IBM i operating system and prevent PAG (Process Access Group) "holes" this program has an "open" and "close" option that can be used **before** LANSA is invoked (e.g.: during system sign on). The "open" and "close" option uses the LANSA convention of CALL M@EXCHDS (X'00') to open and CALL M@EXCHDS (X'FF') to close. Obviously the close operation clears the exchange area. Do **not** call M@EXCHDS from within LANSA RDML functions. These types of calls are **not** required to actually use this facility, only to OPTIMIZE its use.
- Using RCV_DS(*EXCHANGE) is less efficient than using a normal RCV_DS(.....) parameter, but more efficient than using an EXCHANGE command with many fields.

RCV_LIST

Allows up to 20 different working list names to be specified. The following points should be noted when using this parameter:

Each working list specified must be a defined working list within the function. FUNCTION OPTIONS(*DIRECT) must be specified when this parameter is used.

A function receiving a working list will be flagged as being not accessible from the main menu when it is compiled. It will have to be called from another function passing in the correct working lists, rather than from a menu.

The working lists must have been defined with the same attributes in both the called and calling function otherwise errors could occur.

It is also important to note that the order in which the working lists are specified on the PASS_LST parameter of the calling function and the order in which they are specified on the RCV_LIST of the called function is significant - the working lists must appear in the same order in the called and calling functions, otherwise errors could occur.

TRIGGER

is used to specify that this function is to act as a "trigger" for a data dictionary field or a database file.

*NONE, which is the default value, indicates that this function is not a trigger

function.

*FIELD indicates that this function is to act as a data dictionary level trigger. The associated data dictionary field name must also be specified in this parameter.

*FILE indicates that this function is to act as a database level trigger. The associated database file name must also be specified in this parameter. The file specified must be a physical file.

For further details, refer to [Triggers](#).

When a function is defined as a trigger function you **must** follow these guidelines:

- The parameter RCV_LIST(#TRIG_LIST) must be used.
- The parameter RCV_DS must not be used.
- Option *DIRECT must also be used.
- Options xxx_SYSTEM_VARIABLE or xxx_FIELD_VALIDATE must not be used.
- The list #TRIG_LIST must be defined by a DEF_LIST command as DEF_LIST NAME(#TRIG_LIST) TYPE(*WORKING) ENTRYS(2) and must not include any fields in the FIELDS parameter. The required fields will be automatically added.
- No DISPLAY, REQUEST or POP_UP commands may be used. This is a deliberately imposed design/usage constraint that may be removed in later versions.
- No CALL can exist to another process/function. This is a deliberately imposed design/usage constraint that may be removed in later versions.
- If the Built-In Function CALL_SERVER_FUNCTION is used to call a function via SuperServer, you must not pass the list #TRIG_LIST to the server function.
- Trigger functions cannot be defined within an action bar process. This is not to say that they cannot be referenced from within an action bar, it just means that a trigger function cannot be defined as part of a process that is of action bar type.
- The associated process must not have any parameters.
- The exchange list may not be used. This is a deliberately imposed design/usage constraint imposed to enforce insulated and modular design and use of trigger functions.

When a function is defined as a trigger function you **should** follow these guidelines in most situations:

- Understand how triggers are defined and how they should be used by also reading [Triggers](#).
- Use options *NOMESSAGES and *MLOPTIMIZE.
- Options *HEAVYUSAGE and *DBOPTIMIZE may also be considered.
- Do not directly or indirectly access the database file that the trigger is, or will be, linked to.
- Where triggers are heavily and constantly invoked avoid resource intensive operations. Such operations will slow down access to the associated file.
- Recursive implementations may be defined, but will fail to execute correctly. For instance a field trigger function invoked during an insert to file A could attempt to insert data into file B, possibly causing itself to be invoked in a recursive situation, and thus to fail.

7.49.2 FUNCTION Examples

Example 1: Set the possible options for a batch subroutine RDML program:

```
FUNCTION OPTIONS(*NOMESSAGES *HEAVYUSAGE)
```

Example 2: Set possible options for an interactive "pop up selector" RDML subroutine:

```
FUNCTION OPTIONS(*NOMESSAGES *DEFERWRITE)
```

Example 3: Receive data structure DATAFILE into this function:

```
FUNCTION OPTIONS(*DIRECT) RCV_DS(DATAFILE)
```

This requires a file called DATAFILE to be defined to LANSAP. The real fields on this file can be received as a data structure from another function. A dummy field on the end of the data structure would avoid the need to recompile each function referencing the data structure each time fields are added within the data structure. As long as the length of the data structure remains consistent, the functions which use it would not require recompilation (unless the existing fields are changed in length, position or type within the data structure).

Example 4: Receive working list #ORDERLINE into this function

```
FUNCTION OPTIONS(*DIRECT) RCV_LIST(#ORDERLINE)  
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
```

7.50 GET_ENTRY

The GET_ENTRY command is used to retrieve an individual entry from a list. The list may be a **browse** list (used for displaying information at a workstation) or a **working** list (used to store information within a program).

Refer to the DEF_LIST command for more details of lists and list processing.

Also See

[7.50.1 GET_ENTRY Parameters](#)

[7.50.2 GET_ENTRY Examples](#)

Required

*GET_ENTRY ---- NUMBER -----
- numeric value or field name ---->*

Optional

*>--- FROM_LIST ---- *FIRST ----->
list name*

*>-- RET_STATUS --- *STATUS -----|
field name*

7.50.1 GET_ENTRY Parameters

NUMBER

FROM_LIST

RET_STATUS

NUMBER

Specifies a numeric literal or numeric field that indicates the entry number of the list entry that is to be retrieved. As each entry is added to a list by the ADD_ENTRY command it is assigned a number that identifies it. List entries are numbered from 1 (first entry number) to 9999 (maximum possible last entry number) sequentially. By specifying a list entry number it is possible to retrieve an individual list entry.

FROM_LIST

Specifies the name of the list from which the entry should be retrieved.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which may be a browse or a working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

RET_STATUS

Specifies the name of a field that is to receive the "return code" that results from the GET_ENTRY command.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

The GET_ENTRY command only returns 2 possible status codes. These are "OK" (the entry was successfully retrieved) or "NR" (the entry was not found).

7.50.2 GET_ENTRY Examples

Example 1: Retrieve entry number 5 from a list named #ORDERLINE:

```
GET_ENTRY NUMBER(5) FROM_LIST(#ORDERLINE)
```

Example 2: Retrieve entries 7 through 42 from an existing list named #ORDERLINE and increase the value of field #QUANTITY by 10 percent.

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA  
DEFINE FIELD(#ENTRY) TYPE(*DEC) LENGTH(7) DECIMALS(0)
```

```
CHANGE FIELD(#ENTRY) TO(7)  
DOWHILE COND('#ENTRY *LE 42')  
GET_ENTRY NUMBER(#ENTRY) FROM_LIST(#ORDERLINE)  
CHANGE FIELD(#QUANTITY) TO('#QUANTITY * 1.1')  
UPD_ENTRY IN_LIST(#ORDERLINE)  
CHANGE FIELD(#ENTRY) TO('#ENTRY + 1')  
ENDWHILE
```

7.51 GOTO

The GOTO command is used to transfer control to another command in the same function. The command which is to receive control is identified by the label associated with it.

Also See

[7.51.1 GOTO Parameters](#)

[7.51.2 GOTO Examples](#)

Required

GOTO ----- *LABEL* ----- *label* -----|

7.51.1 GOTO Parameters

[LABEL](#)

LABEL

Specifies the label of the command which is to receive control. The label specified in this parameter must be the label of one and only one other command in the function.

7.51.2 GOTO Examples

Example 1: If field #X is less than 10 then transfer control to label L10:

```
IF COND('#X *LT 10')
GOTO LABEL(L10)
ENDIF
```

Example 2: If field #X is less than 10 then transfer control to label L10: else transfer control to label L11:

```
IF COND('#X *LT 10')
GOTO LABEL(L10)
ELSE
GOTO LABEL(L11)
ENDIF
```

7.52 GROUP_BY

The GROUP_BY command is used to group a number of fields together under a common name.

Once fields have been grouped under a "group name" the fields can be referenced as a group by using the group name.

Also See

[7.52.1 GROUP_BY Parameters](#)

[7.52.2 GROUP_BY Comments / Warnings](#)

[7.52.3 GROUP_BY Examples](#)

[7.52.4 GROUP_BY Examples of Expandable Groups](#)

Required

```
GROUP_BY ----- NAME ----- group name -----  
----->
```

```
>-- FIELDS ----- field name field attributes -|  
|           |           ||  
|           --- 7 max ----- |  
|*INCLUDING           |  
|*EXCLUDING           |  
| expandable group     |  
|-- 1000 max for RDMLX-----|  
-- 100 max for RDML -----
```

7.52.1 GROUP_BY Parameters

FIELDS

NAME

NAME

Specifies the name by which the group of fields is to be known. The name specified must start with a # (like field names) and must not be the same as any field defined in this function or the LANSAs data dictionary. Start the name of the group with the prefix "#XG_" if the group is to be an expandable group. For more details about these special types of groups, refer to [Expandable Groups](#).

FIELDS

Specifies the field(s) that are to be in this group. Optionally the field names can be accompanied by field attributes. Refer to [Formats, Values and Codes](#) for further information about specifying field names and field attributes.

The following special values may only be used in expandable groups:

- Expandable group name, in the field list, causes the fields assembled under the group name entry to be expanded and included in the field list instead of the group itself.
- *EXCLUDING in the field list causes the field(s) following this special value to be excluded (if present) from the field list. That is, the field list is switched to exclusion mode.
- *INCLUDING in the field list causes the field to return to inclusion mode. This is required if additional fields need to be added to the field list after an *EXCLUDING entry earlier in the list caused the list to be in exclusion mode.

7.52.2 GROUP_BY Comments / Warnings

A field can be specified in multiple groups.

7.52.3 GROUP_BY Examples

Example 1: Compare the following RDML program:

```
L1: REQUEST FIELDS(#A #B #C #D #E #F #G #H #I)
    INSERT FIELDS(#A #B #C #D #E #F #G #H #I) TO_FILE(TESTFILE)
    GOTO LABEL(L1)
```

with the identical RDML program:

```
GROUP_BY NAME(#GROUP) FIELDS(#A #B #C #D #E #F #G #H #I)
L1: REQUEST FIELDS(#GROUP)
    INSERT FIELDS(#GROUP) TO_FILE(TESTFILE)
    GOTO LABEL(L1)
```

Now if fields #J -> #Z were added to file TESTFILE and this function had to be changed to use these new fields, which function would be the easiest to change?

7.52.4 GROUP_BY Examples of Expandable Groups

Example 1: Groups assembled from other groups:

```
GROUP_BY NAME(#XG_GRP1) FIELDS(#A #B #C #D)
GROUP_BY NAME(#XG_GRP2) FIELDS(#E #F #G #H)
GROUP_BY NAME(#XG_GRP3) FIELDS(#XG_GRP1 #XG_GRP2)
```

is equivalent to writing:

```
GROUP_BY NAME(#XG_GRP3) FIELDS(#A #B #C #D #E #F #G #H)
```

Example 2: Groups assembled from other groups which share common fields:

```
GROUP_BY NAME(#XG_GRP1) FIELDS(#A #B #C #D)
GROUP_BY NAME(#XG_GRP2) FIELDS(#C #D #E #F)
GROUP_BY NAME(#XG_GRP3) FIELDS(#XG_GRP1 #XG_GRP2)
```

is equivalent to writing:

```
GROUP_BY NAME(#XG_GRP3) FIELDS(#A #B #C #D #E #F)
```

When the field list is expanded, the fields from group #XG_GRP2 which are already in the field list are not added again.

Example 3: Mixing expandable groups and individual fields in the field list:

```
GROUP_BY NAME(#XG_GRP1) FIELDS(#A #B #C)
GROUP_BY NAME(#XG_GRP2) FIELDS(#E #F #G)
GROUP_BY NAME(#XG_GRP3) FIELDS(#XG_GRP1 #D #XG_GRP2 #H)
```

is equivalent to writing:

```
GROUP_BY NAME(#XG_GRP3) FIELDS(#A #B #C #D #E #F #G #H)
```

Example 4: Using *EXCLUDING and *INCLUDING special values in the field list:

```
GROUP_BY NAME(#XG_GRP1) FIELDS(#A #B #C #D #E)
GROUP_BY NAME(#XG_GRP2) FIELDS(#F #G #H #I #J)
GROUP_BY NAME(#XG_GRP3) FIELDS(#XG_GRP1 *EXCLUDING #D :
```

is equivalent to writing:

GROUP_BY NAME(#XG_GRP3) FIELDS(#A #B #C #F #G #H #I)

7.53 IF

The IF command is used to test the truth of a condition and then execute certain RDML commands only if the condition is true.

By using an ELSE command in conjunction with an IF command it is possible to nominate the RDML commands to execute if the condition is true and the commands to execute if the condition is not true.

An IF command is always used in conjunction with an ENDIF command and optionally may be used in conjunction with an ELSE command. Refer to these commands for more information.

Also See

[7.53.1 IF Parameters](#)

[7.53.2 IF Examples](#)

Required

IF ----- 'condition' -----|

7.53.1 IF Parameters

COND

COND

Specifies the condition that is to be evaluated to test the "truth" of the IF condition. For more details, refer to [Specifying Conditions and Expressions](#).

7.53.2 IF Examples

Example 1: If field #I is greater than 10 issue a message indicating this, else issue a message indicating it is less than or equal to 10:

```
IF    COND('#I *GT 10')
MESSAGE MSGTXT('#I is greater than 10')
ELSE
MESSAGE MSGTXT('#I is less than or equal to 10')
ENDIF
```

Example 2: Execute a certain series of commands if #QUANTITY is less than 10 and #MEASURE is greater than 42.67, else execute a different series of commands:

```
IF    COND('( #QUANTITY *LT 10) *AND (#MEASURE *GT 42.67)')
* << commands to execute when condition is true >>
ELSE
* << commands to execute when condition is false >>
ENDIF
```

7.54 IF_ERROR

The IF_ERROR command is a hybrid version of the normal IF command. It allows for the "error condition" inside a validation block to be checked.

For more information about the raising of the "error condition" inside a validation block refer to the ENDCHECK command.

An optional error message may be issued when the "error condition is raised".

Also See

[7.54.1 IF_ERROR Parameters](#)

[7.54.2 IF_ERROR Examples](#)

Optional

```
IF_ERROR ----- MSGTXT ----- *NONE -----  
----->  
        message text  
  
>-- MSGID ----- *NONE ----->  
        message identifier  
  
>-- MSGF ----- DC@M01 . *LIBL -----  
>  
        message file . library name  
  
>-- MSGDTA ----- substitution variables -----|  
        | expandable group expression |  
        ----- 20 max -----
```

7.54.1 IF_ERROR Parameters

MSGDTA

MSGF

MSGID

MSGTXT

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing

blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this:

```
MSGDTA('BOLTS ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA("BOLTS " #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.54.2 IF_ERROR Examples

There are no examples for the IF_ERROR Command

7.55 IF_KEY

The IF_KEY command is a hybrid version of the normal IF command. It allows the last key used at a workstation to be tested.

For more information about the keys that are enabled by commands that use a workstation refer to the DISPLAY, REQUEST, POP_UP and BROWSE commands.

Also See

[7.55.1 IF_KEY Parameters](#)

[7.55.2 IF_KEY Comments / Warnings](#)

[7.55.3 IF_KEY Examples](#)

Required

```
IF_KEY ----- WAS ----- *ENTER -----
-|
| *EXIT      |
| *ADD       |
| *CHANGE   |
| *DELETE   |
| *PROMPT   |
| *CANCEL   |
| *EXITHIGH |
| *EXITLOW  |
| *ROLLUP   |
| *ROLLDOWN |
| *USERKEY1 |
| *USERKEY2 |
| *USERKEY3 |
| *USERKEY4 |
| *USERKEY5 |
| *BUTTONnn |
|           |
|----- 10 maximum -----
```

NOTE : nn = "1 " to "20"

7.55.1 IF_KEY Parameters

WAS

WAS

Specifies the workstation keys that will make the IF condition true. If any one of the keys specified matches the last key used at the workstation then the IF statement is deemed to be "true" (ie: the condition is an OR condition).

7.55.2 IF_KEY Comments / Warnings

- The IF_KEY command should only be used after a command that uses the workstation such as DISPLAY, REQUEST, POP_UP or BROWSE has been executed. Using an IF_KEY command before any interaction with a workstation will produce unpredictable results.
- The value *PROMPT allows tests to be made for the use of the prompt key. However in most RDML programs the prompt key is handled automatically because the associated parameter is coded as PROMPT_KEY(*YES *AUTO). To avoid having the prompt key handled automatically code PROMPT_KEY(*YES *NEXT) or PROMPT_KEY(*YES L10) where L10 is a command label. In these cases the prompt key is deemed to be handled by the RDML program, and thus the IF_KEY command can be used to test for its use.
- The test values *CANCEL, *EXITHIGH and *EXITLOW are provided for complete compatibility with SAA/CUA function key assignments. Do not use these values in non-SAA/CUA applications.
- *CANCEL is functionally identical to *MENU.
- When testing for *EXITHIGH and *EXITLOW ensure that the associated DISPLAY, REQUEST or POP_UP command correctly enables either the exit high key or the exit low key via the EXIT_KEY parameter. It is not possible to enable a high and low exit on the one DISPLAY, REQUEST or POP_UP command.
- The LANSAs dictionary contains a special field called #IO\$KEY which always contains the last function key used during a workstation interaction. This can be tested in an IF or CASE statement like any other field. In some situations using this field in an IF or CASE command is a better solution than using the hybrid IF_KEY command.

An example of using this field follows:

```
DISPLAY USER_KEYS((15 'Purge')(16 'Commit')
(17 'Save') (*ROLLUP 'Up')(*ROLLDOWN 'Down'))
```

```
CASE OF_FIELD(#IO$KEY)
  WHEN VALUE_IS(= "15")
  WHEN VALUE_IS(= "16")
  WHEN VALUE_IS(= "17")
  WHEN VALUE_IS(= "UP")
```

```
      WHEN VALUE_IS(= "DN")
ENDCASE
```

Field #IO\$KEY can be compared for normal function key use by using the function, key number (ie: 01 to 24). However, some other values may occur in #IO\$KEY during an RDML programs execution. These include the following: 'RA' - The record advance / enter key was used 'UP' - The roll up key was used 'DN' - The roll down key was used These values are referred to as AIDS values and are actually part of the i5/05 operating system.

- If a DISPLAY, REQUEST or POP_UP command has been executed that enables the roll up (*ROLLUP) or roll down (*ROLLDOWN) keys, and a browse list is also displayed, control will not actually return to the RDML program until the user rolls off either "end" of the browselist.

For instance if a browse list is filled with 42 entries (assume 15 entries are displayed on each "page" of the display) and then displayed, the roll up key processing is handled automatically by the i5/05 operating system until the 3rd use. On the 3rd use the user has attempted to roll off the "end" of the browselist, thus control is returned to the RDML program. Roll down processing is handled similarly.

7.55.3 IF_KEY Examples

The following example applies to the IF_KEY command.

Use the IF_KEY command to indicate to the user which function key he/she pressed:

```
BEGIN_LOOP
```

```
DISPLAY FIELDS(#ORDNUM #CUSTNUM  
#DATEDUE) EXIT_KEY(*YES L01) MENU_KEY(*YES L01) ADD_KEY(  
(16 'Task2')(17 'Task3') (18 'Task4')(19 'Task5'))
```

```
L01: IF_KEY WAS(*ENTER)
```

```
MESSAGE MSGTXT('The ENTER key was pressed')  
ENDIF
```

```
IF_KEY WAS(*MENU)
```

```
MESSAGE MSGTXT('The MENU key was pressed')  
ENDIF
```

```
IF_KEY WAS(*ADD)
```

```
MESSAGE MSGTXT('The ADD key was pressed')  
ENDIF
```

```
IF_KEY WAS(*USERKEY1)
```

```
MESSAGE MSGTXT('User key 1 (F15) was pressed')  
ENDIF
```

```
IF_KEY WAS(*USERKEY2)
```

```
MESSAGE MSGTXT('User key 2 (F16) was pressed')  
ENDIF
```

```
IF_KEY WAS(*USERKEY3)
```

```
MESSAGE MSGTXT('User key 3 (F17) was pressed')  
ENDIF
```

```
IF_KEY WAS(*USERKEY4)
```

```
MESSAGE MSGTXT('User key 4 (F18) was pressed')  
ENDIF
```

```
IF_KEY WAS(*USERKEY5)
MESSAGE MSGTXT('User key 5 (F19) was pressed')
ENDIF
```

```
END_LOOP
```

For more information about function key use and function key assignments refer to the DISPLAY, REQUEST, POP_UP and BROWSE commands.

7.56 IF_MODE

The IF_MODE command is a hybrid version of the normal IF command. It allows the current screen "mode" to be tested.

For more information, refer to [Screen Modes and Mode Sensitive Commands](#).

Also See

[7.56.1 IF_MODE Parameters](#)

[7.56.2 IF_MODE Examples](#)

Optional

```
IF_MODE ----- IS ----- *DISPLAY -----  
-|  
    *ADD  
    *CHANGE  
    *DELETE
```

7.56.1 IF_MODE Parameters

IS

IS

Specifies the screen mode that is to be tested for. Allowable values are *DISPLAY, *ADD, *CHANGE and *DELETE. For more details of screen processing modes and the standard screen processing logic, refer to [Screen Modes and Mode Sensitive Commands](#).

7.56.2 IF_MODE Examples

The following example applies to the IF_MODE command.

Create a simple inquire/add/update/delete function on a file called NAMES. Use the IF_MODE command to test the mode that the screen was in at the time the enter key was pressed:

```
GROUP_BY NAME(#NAMEINFO) FIELDS(#CUSTNO #NAME #ADD1

L10: CHANGE FIELD(#CUSTNO) TO(*DEFAULT)

MESSAGE MSGTXT('Specify customer to review or use ADD key to add

L15: SET_MODE TO(*DISPLAY)
REQUEST FIELDS(#CUSTNO) ADD_KEY(*YES)
*
* Add a new customer to the file .....
*
IF_MODE IS(*ADD)
    REQUEST FIELDS(#NAMEINFO)
    INSERT FIELDS(#NAMEINFO) TO_FILE(NAMES)
*
* Else review / change / delete an existing customer
*
ELSE
    FETCH FIELDS(#NAMEINFO) FROM_FILE(NAMES) WITH_KEY

    DISPLAY FIELDS(#NAMEINFO) CHANGE_KEY(*YES) DELETE_I

    IF_MODE IS(*CHANGE)
    UPDATE FIELDS(#NAMEINFO) IN_FILE(NAMES)
    ENDIF

    IF_MODE IS(*DELETE)
    DELETE FROM_FILE(NAMES)
    ENDIF

ENDIF
*
```

* Go back and request next customer

*

GOTO LABEL(L10)

7.57 IF_NULL

The IF_NULL command is a hybrid version of the normal IF command. It allows one or more field(s) to be tested for the "null" values.

For a definition of the *NULL value for each of the field types, refer to [7.9.1 CHANGE Parameters](#).

Also See

[7.57.1 IF_NULL Parameters](#)

[7.57.2 IF_NULL Comments / Warnings](#)

[7.57.3 IF_NULL Examples](#)

Required

```
IF_NULL ----- FIELD ----- field name -----  
|  
|           | expandable group expression |  
|           |  
|           ----- 100 max -----
```

7.57.1 IF_NULL Parameters

FIELD

FIELD

Specifies the names of the field(s) or group(s) (refer to the GROUP_BY command) that are to be checked for the "null" condition. An expandable group expression is allowed in this parameter.

Note that the relationship between the field(s) in the list is an AND relationship. Thus all fields specified must have null values to satisfy the condition.

7.57.2 IF_NULL Comments / Warnings

Fields that are SQL Null will fail the IF_NULL check.

If you only need to check for SQL Null, use code like the following:

```
If '#FIELD.IsSqlNull'  
...  
Endif
```

If you want a condition to return True if the field is SQL Null or Null, use code like the following:

```
If '#FIELD.IsSqlNull *OR #FIELD.IsNull'  
...  
Endif
```

Alternatively, you can compare against the null value for the field. The following If example returns True if #Field is *SQLNULL or *NULL (assuming #Field is numeric) .

```
If '#FIELD *EQ *ZERO'  
...  
Endif
```

7.57.3 IF_NULL Examples

Example 1: If field #A and #C are numeric and field #B is alphanumeric then the following IF conditions are all identical (assuming none of the fields are SQL Null):

```
IF_NULL FIELD(#A #B #C)
```

is identical to:

```
GROUP_BY NAME(#GROUP) FIELDS(#A #B #C)  
IF_NULL FIELD(#GROUP)
```

which is identical to:

```
IF COND('#A = 0) AND (#B = *BLANKS) AND (#C = 0)')
```

which is identical to:

```
IF COND('#A = *ZERO) AND (#B = " ") AND (#C = *ZERO)')
```

Example 2: Request that the user supplies values for fields #A, #B and #C. If no values are specified assume that the user wants to end the function and re-display the process's main menu:

```
GROUP_BY NAME(#GROUP) FIELDS(#A #B #C)
```

```
CHANGE FIELD(#GROUP) TO(*NULL)  
REQUEST FIELDS(#GROUP)
```

```
IF_NULL FIELD(#GROUP)  
MENU MSGTXT('Since no data entered, end of function assumed')  
ENDIF
```

7.58 IF_STATUS

The IF_STATUS command is a hybrid version of the normal IF command. It allows the status of the last I/O command to be tested without specific references to the I/O return codes.

For more information, refer to *I/O Return Codes*.

Also See

[7.58.1 IF_STATUS Parameters](#)

[7.58.2 IF_STATUS Comments / Warnings](#)

[7.58.3 IF_STATUS Examples](#)

Optional

```
IF_STATUS ---- IS ----- *OKAY -----
```

>

```
*ERROR  
*VALERROR  
*NORECORD  
*ENDFILE  
*BEGINFILE  
*EQUALKEY  
*NOTEQUALKEY
```

```
>--- IS_NOT ----- *OKAY -----|
```

```
*ERROR  
*VALERROR  
*NORECORD  
*ENDFILE  
*BEGINFILE  
*EQUALKEY  
*NOTEQUALKEY
```

7.58.1 IF_STATUS Parameters

IS

IS_NOT

IS

Specifies the I/O return codes that will make the IF condition true. If any one of the I/O codes matches the I/O return code of the last I/O operation then the IF statement is deemed to be "true".

IS_NOT

Specifies the I/O return codes that will make the IF condition true in negation. If any one of the I/O codes does not match the I/O return code of the last I/O operation then the IF statement is deemed to be "true".

7.58.2 IF_STATUS Comments / Warnings

- The following table matches the values specified on the IS or IS_NOT parameters with the actual I/O return codes. For more information, refer to [I/O Return Codes](#):

Parameter Value	I/O Return Code
------------------------	------------------------

*OKAY	'OK'
-------	------

*VALERROR	'VE'
-----------	------

*ERROR	'ER'
--------	------

*NORECORD	'NR'
-----------	------

*ENDFILE	'EF'
----------	------

*BEGINFILE	'BF'
------------	------

*NOTEQUALKEY	'NE'
--------------	------

*EQUALKEY	'EQ'
-----------	------

- IF_STATUS does NOT check the value of #IO\$STS. IF_STATUS simply returns the status of the last database I/O command. In many cases this will be the same as IF COND(#IO\$STS *EQ {Some Value}). However it should be noted that in the following cases these statements will be different:
 - When #IO\$STS is being set through an Output mapping of a MthRoutine.
 - When #IO\$STS is being set through an Output mapping of a SubRoutine.
 - When #IO\$STS is being used in the TO_GET parameter of a USE command.
 - When #IO\$STS is EXCHANGE'ed from another program/function.

7.58.3 IF_STATUS Examples

Example 1: Use the IF_STATUS command to trap a record not found condition and abort the function with an error:

```
FETCH  FIELDS(#NAMEINFO) FROM_FILE(NAMES) WITH_KEY(#CU
IF_STATUS IS_NOT(*OKAY)
ABORT  MSGTXT('Customer name details not found')
ENDIF
```

which is identical to the following:

```
FETCH  FIELDS(#NAMEINFO) FROM_FILE(NAMES) WITH_KEY(#CU
IF    COND('#IO$STS *NE OK')
ABORT  MSGTXT('Customer name details not found')
ENDIF
```

Example 2: Modify the previous example to display the details (if found) else to abort the function:

```
FETCH  FIELDS(#NAMEINFO) FROM_FILE(NAMES) WITH_KEY(#CU
IF_STATUS IS(*OKAY)
DISPLAY FIELDS(#NAMEINFO)
ELSE
ABORT  MSGTXT('Customer name details not found')
ENDIF
```

which is identical to the following:

```
FETCH  FIELDS(#NAMEINFO) FROM_FILE(NAMES) WITH_KEY(#CU
IF    COND('#IO$STS = OK')
DISPLAY FIELDS(#NAMEINFO)
ELSE
ABORT  MSGTXT('Customer name details not found')
ENDIF
```

7.59 INCLUDE

The INCLUDE command is used to include RDML from another function.

Portability Considerations Refer to [7.59.2 INCLUDE Comments / Warnings](#).

Also See

[7.59.1 INCLUDE Parameters](#)

[7.59.2 INCLUDE Comments / Warnings](#)

[7.59.3 INCLUDE Examples](#)

```
INCLUDE ----- PROCESS----- process name -----  
----->  
                *DIRECT  
  
>-- FUNCTION ----- function name -----
```

7.59.1 INCLUDE Parameters

FUNCTION

PROCESS

PROCESS

Specifies the name of the LANSAs process from which RDML is to be included. This parameter must be specified.

If the function name from which RDML is to be included is unique in the partition, simply specify the name *DIRECT in this parameter, in place of the actual process name, and then nominate the function name in the FUNCTION parameter.

FUNCTION

Specifies the name of the LANSAs function from which RDML is to be included. This parameter must be specified.

7.59.2 INCLUDE Comments / Warnings

This note applies only to IBM i:

If PROCESS(*DIRECT) is specified and the function name is not unique in the partition, then the RDML code is included from the function of that name in the process name that is highest in the normal collating sequence. For example, if function F is in both process B and process C, then INCLUDE PROCESS(*DIRECT) FUNCTION(F) will cause RDML to be included from function F in process B. If function F is later created in process A, a subsequent compile will cause RDML to be included from function F in process A.

This note applies to Visual LANSA:

If PROCESS(*DIRECT) is specified the function name must be unique within the partition.

If process B and process C both have the same function named function F, the function F process will not compile on both process B and process C. This would cause the following error message to be displayed:

Function name must be unique within the partition to use OPTIONS(*DIRECT).

However, if OPTIONS(*DIRECT) is NOT used, this error message is displayed:

To generate C code you must use FUNCTION OPTIONS(*DIRECT) in this function.

Therefore, it is not possible to use the same function name in the partition.

INCLUDE Comments / Warnings (all LANSA)

- INCLUDE cannot be embedded in RDML that is already included.
- Import / export, checkin / checkout and the deployment tool will not automatically cause functions that are included to also be processed.
- Impact Analysis will not automatically associate included functions with the functions that INCLUDE them via the related object search, or vice versa. A profile search scanning for the function name in RDML code will allow the where used capability.
- The total number of lines of RDML code in a function in IBM i cannot be greater than 4096 counting the included RDML. This restriction does not apply to a function in Visual LANSA.
- Included screens (DISPLAY, REQUEST, POP_UP, supporting DEF_LIST &

GROUP_BY) & reports (DEF_LINE, DEF_HEAD, DEF_FOOT, DEF_BREAK) cannot be painted.

- Included screens & reports are not recommended.
- The full function checker reports any errors in included RDML against the relevant INCLUDE RDML command.
- The INCLUDE RDML command is not valid in components.
- Print of functions does not expand included RDML commands.
- Included RDML cannot be debugged on LANSAs for IBM i
- Run time errors in included RDML will be reported against the relevant INCLUDE RDML command

7.59.3 INCLUDE Examples

[Include common executable code in functions](#)

[Include common declarations in Functions](#)

Include common executable code in functions

A common set of code exists that needs to be executed from many functions.

FUNCA RDML

```
SUBROUTINE NAME(SUB1)
  *<<common set of code>>
ENDROUTINE
```

FUNCB RDML

```
EXECUTE SUBROUTINE(SUB1)

INCLUDE PROCESS(PROCA) FUNCTION(FUNCA)
```

As an alternative, you could place the common code in its own function and call it where it is needed, or copy it in to each function that needs it. Using the INCLUDE command is the recommended approach.

Include common declarations in Functions

One function calls another function, passing a working list. The working list needs to be defined exactly the same in both functions.

FUNC1 RDML

```
DEF_LIST NAME(#WRKLIST) FIELDS(<<fields needed>>) TYPE(*WORK
```

FUNC2 RDML

```
INCLUDE PROCESS(*DIRECT) FUNCTION(FUNC1)

CALL PROCESS(*DIRECT) FUNCTION(FUNC3) PASS_LST(#WRKLIST)
```

FUNC3 RDML

```
FUNCTION OPTIONS(*DIRECT) RCV_LIST(#WRKLIST)
INCLUDE PROCESS(*DIRECT) FUNCTION(FUNC1)
```

Alternatives would be either to define the working list in one function then copy

it into the other function that needs it, or to include it in the functions that need it.

7.60 INSERT

The INSERT command is used to insert fields into a new record in a file.

Portability Considerations Refer to parameters: [AUTOCOMMIT](#) and [TO_FILE](#) .

Also See

[7.60.1 INSERT Parameters](#)

[7.60.2 INSERT Comments / Warnings](#)

[7.60.3 INSERT Examples](#)

Required

INSERT ----- *FIELDS* ----- *field name field attributes* --
->

```
|          |          ||
|          --- 7 max ----- |
|*ALL          |
|*ALL_REAL          |
|*ALL_VIRT          |
|*INCLUDING          |
|*EXCLUDING          |
|expandable group          |
|          |
|----- 1000 max for RDMLX-----|
|----- 100 max for RDML -----|
```

>-- *TO_FILE* ----- *file name . *FIRST* ----->
 library name

Optional

>-- *IO_STATUS* ---- **STATUS* ----->
 field name

>-- *IO_ERROR* ----- **ABORT* -----

>

**NEXT*
**RETURN*
label

>-- VAL_ERROR ---- *LASTDIS -----
>

**NEXT*
**RETURN*
label

>-- ISSUE_MSG ---- *NO ----->
**YES*

>-- RETURN_RRN --- *NONE -----
->

>-- CHECK_ONLY --- *NO -----
>
**YES*

>-- AUTOCOMMIT --- *FILEDEF -----
---|
**YES*
**NO*

7.60.1 INSERT Parameters

AUTOCOMMIT

CHECK_ONLY

FIELDS

IO_ERROR

IO_STATUS

ISSUE_MSG

RETURN_RRN

TO_FILE

VAL_ERROR

FIELDS

Specifies either the field(s) that are to be inserted into the file or the name of a group that specifies the field(s) to be inserted.

Alternatively, an expandable group expression can be entered in this parameter. For more details, refer to [Expandable Groups](#). The following special values can be used:

- *ALL, specifies that all fields from the currently active file be inserted.
- *ALL_REAL, specifies that all real fields from the currently active file be inserted.
- *ALL_VIRT, specifies that all virtual fields from the currently active file be inserted.
- *EXCLUDING, specifies that fields following this special value must be excluded from the field list.
- *INCLUDING, specifies that fields following this special value must be included in the field list. This special value is only required after an *EXCLUDING entry has caused the field list to be in exclusion mode.

Note: When all fields are inserted from a logical file maintained by OTHER, all the fields from the based-on physical file are included in the field list.

It is strongly recommended that the special values *ALL, *ALL_REAL or *ALL_VIRT in parameter FIELDS be used sparingly and only when strictly required. Inserting fields which are not needed invalidates cross-reference

details (shows fields which are not used in the function) and increases the Crude Entity Complexity Rating of the function pointlessly.

TO_FILE

Refer to [Specifying File Names in I/O commands](#).

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

For values, refer to [I/O Return Codes](#).

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples: a file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

*ABORT, the default value, indicates the function will abort with error messages that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command. The purpose of *NEXT is to permit you to handle error messages in the RDML, and then ABORT, rather than use the default ABORT. (It is possible for processing to continue for LANSAs on IBM i and Visual LANSAs, but this is NOT a recommended way to use LANSAs.)

ER returned from a database operation is a fatal error and LANSAs does not expect processing to continue. The IO Module is reset and further IO will be as if no previous IO on that file had occurred. Thus you must not make any presumptions as to the state of the file. For example, the last record read will not be set. A special case of an IO_ERROR is when a trigger function is coded to return ER in TRIG_RETC. The above description applies to this case as well. Therefore, LANSAs recommends that you do NOT use a return code of ER from a trigger function to cause anything but an ABORT or EXIT to occur before any

further IO is performed.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

VAL_ERROR

Specifies the action to be taken if a validation error was detected by the command.

A validation error occurs when information that is to be added, updated or deleted from the file does not pass the FILE or DICTIONARY level validation checks associated with fields in the file.

The default value *LASTDIS specifies that control will be passed back to the last display screen used. The fields that failed the associated validation checks will be displayed in reverse image and the cursor positioned to the first field in error on the screen.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).

When using *LASTDIS the "Last Display" must be at the same level as the database command (INSERT, UPDATE, DELETE, FETCH and SELECT). If they are at different levels e.g. the database command is specified in a SUBROUTINE, but the "Last Display" is a caller routine or the mainline, the function will abort with the appropriate error message(s).

The same does NOT apply to the use of event routines and method routines in Visual LANSA. In these cases, control will be returned to the calling routine. The fields will display in error with messages

returned to the first status bar encountered in the parent chain of forms, or if none exist, the first form with a status bar encountered in the execution stack (for example, a reusable part that inherits from PRIM_OBJT).

ISSUE_MSG

This parameter is redundant. Its value has no effect.

The default value is *NO.

The only other allowable value is *YES.

RETURN_RRN

Specifies the name of a field in which the relative record number of the record inserted should be returned.

Any field nominated in this parameter must be defined within the function or the LANSAs data dictionary and must be numeric.

For further information refer also to [Load Other File](#) in the *Visual LANSAs Developers Guide*.

CHECK_ONLY

Indicates whether the I/O operation should actually be performed or only "simulated" to check whether all file and data dictionary level validation checks can be satisfied when it is actually performed.

*NO, which is the default value, indicates that the I/O operation should be performed in the normal manner.

*YES indicates that the I/O operation should be simulated to verify that all file and data dictionary level checks can be satisfied. The database file involved is not changed in any way when this option is used.

AUTOCOMMIT

This parameter was made redundant in LANSAs release 4.0 at program change level E5.

To use commitment control specify COMMIT and/or ROLLBACK commands in your application.

Generally only COMMIT commands are required.

For the implications of using commitment control on the IBM i, refer to [Commitment Control](#) in the *LANSAs for i User Guide*.

Portability

If using Visual LANSAs, refer to [Commitment Control](#) in the

Considerations *LANSA Application Design Guide.*

7.60.2 INSERT Comments / Warnings

- Where fields are not specified for the new record they will adopt their default values as defined in the data dictionary. For example, if a record in file NAMES contained fields #CUSTNO (customer number), #NAME (customer name), #ADD1 (address line 1) and #POSTCD (post code), then the following command:

```
INSERT FIELDS(#CUSTNO #NAME) TO_FILE(NAMES)
```

would cause #ADD1 to be set to blanks in the new record and #POSTCD to be set to 2000 (if these were the data dictionary defaults for the fields).

- Note that when BLOB or CLOB data is inserted, it should be either *SQLNULL, *NULL or a filename. If a filename, it is assumed that the file exists and that the contents are to be copied into the BLOB or CLOB in the database.
- Any INSERT operation must include at least the "primary key fields" of the file. The primary key fields are specified when the file is set up. If the primary key fields are not specified the INSERT operation will fail with an "insufficient information" error.

For example, if the primary key of file NAMES is #CUSTNO then the following operations will fail:

```
INSERT FIELDS(#NAME #ADD1) TO_FILE(NAMES)
```

```
INSERT FIELDS(#POSTCD) TO_FILE(NAMES)
```

- When an SQL Null field is inserted into a table's database column, one of the following will occur:
 - If the column has a default or automatically generated value defined (in the DBMS, not LANSA), the default value will be inserted, rather than the SQL Null.
 - If the column does not have the NOT NULL constraint, the column is set to SQL Null.
 - If the column does have the NOT NULL constraint, the insert will fail. (This can only occur if the database definition of the column does not match the LANSA definition of the field.)

7.60.3 INSERT Examples

Example 1: Insert fields #CUSTNO, #NAME, #ADDL1 and #POSTCD into a file named CUSTMST that has key:

```
INSERT FIELDS(#CUSTNO #NAME #ADDL1 #POSTCD) TO_FILE(CUSTMST)
```

or identically:

```
GROUP_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDL1 #POSTCD)
INSERT FIELDS(#CUSTOMER) TO_FILE(CUSTMST)
```

Example 2: Request that the user input some customer details. If the customer already exists update the fields, else create a new customer record:

```
GROUP_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDL1 #POSTCD)
```

```
REQUEST FIELDS(#CUSTOMER)
CHECK_FOR IN_FILE(CUSTMST) WITH_KEY(#CUSTNO)
```

```
IF_STATUS IS(*EQUALKEY)
UPDATE FIELDS(#CUSTOMER) IN_FILE(CUSTMST) WITH_KEY(#CUSTOMER)
ELSE
INSERT FIELDS(#CUSTOMER) TO_FILE(CUSTMST)
ENDIF
```

Example 3: Insert all real fields from the currently active version into file CUSTMST:

```
INSERT FIELDS(*ALL_REAL) TO_FILE(CUSTMST)
```

Example 4: Exclude address fields during insertion of a new record into file CUSTMST:

```
GROUP_BY NAME(#XG_ADDR) FIELDS(#ADDL1 #POSTCD)
INSERT FIELDS(*ALL *EXCLUDING
#XG_ADDR) TO_FILE(CUSTMST)
```

7.61 INZ_LIST

The INZ_LIST command is used to initialize a list with a set number of identical entries. All entries will be set to the values the list's fields contain at the time the INZ_LIST command is executed.

The list may be a **browse** list (used for displaying information at a workstation) or a **working** list (used to store information within a program).

The INZ_LIST command is normally used to initialize a list with a number of "null" entries that are to be used for data entry (rather than display) purposes. Refer to the DEF_LIST command for more details of lists and list processing.

Also See

[7.61.1 INZ_LIST Parameters](#)

[7.61.2 INZ_LIST Comments / Warnings](#)

[7.61.3 INZ_LIST Examples](#)

Optional

```
INZ_LIST ----- NAMED ----- *FIRST -----
-->
      list name

>-- NUM_ENTRYS --- 1 ----->
      number of entries

>-- SET_SELECT --- *YES ----->
      *NO

>-- WITH_MODE ---- *CURRENT -----
---|
      *ADD
      *CHANGE
      *DELETE
      *DISPLAY
      field name
```

7.61.1 INZ_LIST Parameters

NAMED

NUM_ENTRIES

SET_SELECT

WITH_MODE

NAMED

Specifies the name of the list which is to be initialized.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which may be a browse or a working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

NUM_ENTRIES

Specifies the number of entries that should be initialized into the list.

The default value is 1. Any other value specified must be in the range 1 to 9999.

SET_SELECT

Specifies whether or not any fields in the list that have special attribute *SELECT should be set to blanks before the new entry is added to the list. Refer to the DEF_LIST command for more details.

This is only valid for browse lists. It is ignored for working lists.

WITH_MODE

Specifies the mode to be set for the entries being initialized into the list. This overrides the mode that has been set by the SET_MODE command (refer to the SET_MODE command).

The default is *CURRENT which uses the current mode that has been set by the SET_MODE command. Other allowable values are *ADD, *CHANGE, *DELETE and *DISPLAY. A user field name may also be specified, and must be alphanumeric with a length of 3, and must contain one of the values "ADD", "CHG", "DLT" or "DIS".

7.61.2 INZ_LIST Comments / Warnings

INZ_LIST is a "mode sensitive" command when being used with a browse list. For details of mode sensitive commands, refer to [Screen Modes and Mode Sensitive Commands](#).

7.61.3 INZ_LIST Examples

Example 1: Initialize a list named #ORDERLINE with 100 "null" entries that are to be used for data entry (input capable on the display):

```
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
CHANGE    FIELD(#ORDERLINE) TO(*NULL)
SET_MODE  TO(*ADD)
INZ_LIST  NAMED(#ORDERLINE) NUM_ENTRYYS(100)
```

Example 2: Use the list created in example 1 to perform multiple line data entry for an order lines file:

```
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
CHANGE    FIELD(#ORDERLINE) TO(*NULL)
SET_MODE  TO(*ADD)
INZ_LIST  NAMED(#ORDERLINE) NUM_ENTRYYS(100)

REQUEST  FIELDS(#ORDNUM) BROWSELIST(#ORDERLINE)

SELECTLIST NAMED(#ORDERLINE) GET_ENTRYYS(*NOTNULL)
INSERT    FIELDS(#ORDERLINE) TO_FILE(ORDLIN)
ENDSELECT
```

7.62 KEEP_AVG

The KEEP_AVG command is used to keep the average of field(s) in another field.

Note: Full RDMLX Fields cannot be used with this command.

The KEEP_AVG command is only valid within SELECT or SELECTLIST command loops, because the processing logic used is implicitly linked to the SELECT / SELECTLIST loop logic.

Normally the KEEP_AVG command is entered directly after the SELECT or SELECTLIST command. However there are specific exceptions to this. Refer to the examples section for more details.

Also See

[7.62.1 KEEP_AVG Parameters](#)

[7.62.3 KEEP_AVG Examples](#)

[7.62.2 KEEP_AVG Comments / Warnings](#)

Required

KEEP_AVG --- OF_FIELD ---- list of field names -----
----->

| expandable group expression |
----- -- 50 max - -----

IN_FIELD ---- field name ----->

Optional

*BY_FIELD ---- *SELECTED -----|*
list of field names
| expandable group expression |
----- 20 max -----

7.62.1 KEEP_AVG Parameters

BY_FIELD

IN_FIELD

OF_FIELD

OF_FIELD

Specifies from 1 to 50 fields that are to be averaged by the KEEP_AVG command.

Fields specified in this parameter must be defined in the LANSAs data dictionary or defined within this function by a DEFINE command. Expandable group expressions are valid in this parameter.

The fields specified must also be numeric.

IN_FIELD

Specifies the field that is to hold the result of the averaging of the field(s) specified in the OF_FIELD parameter.

The field specified in this parameter must be numeric and must be defined in the LANSAs data dictionary or defined in this function with a DEFINE command.

This field is reset at the beginning of each SELECT/SELECTLIST loop.

BY_FIELD

Specifies the condition under which the fields are to be averaged.

*SELECTED, which is the default value indicates, that the averaging should continue until the SELECT/SELECTLIST loop terminates (ie: all selected information).

Otherwise, specify a list of 1 to 20 field names (alternatively, enter an expandable group expression). The averaging continues until **one or more** of the fields in the list changes value. When one or more of the fields changes value the "accumulator".is reset to its null value and a new averaging cycle is started.

The "accumulator" is the work field calculating the average.

See the examples section for more details of how this parameter is used.

7.62.2 KEEP_AVG Comments / Warnings

Refer to the [7.63.2 KEEP_COUNT Comments / Warnings](#) which are also applicable to KEEP_AVG.

7.62.3 KEEP_AVG Examples

Example 1: A sales history file contains details of a company name (#COMPANY), a division name (#DIVNAM) and a sales value (#SALES). Print all sales details and print total and average sales details by division, company and report (grand total):

```
DEF_LINE NAME(#DETAIL) FIELDS(#COMPANY #DIVNAM #SALES)
DEF_BREAK NAME(#SUBDIV) FIELDS(#DIVTOT #DIVAVG) TRIGGER
DEF_BREAK NAME(#SUBCOM) FIELDS(#COMTOT #COMAVG) TRIGGER
DEF_BREAK NAME(#GRAND) FIELDS(#GRDTOT #GRDAVG)
SELECT FIELDS(#DETAIL) FROM_FILE(SALEHIST)
KEEP_TOTAL OF_FIELD(#SALES) IN_FIELD(#DIVTOT) BY_FIELD(#COMPANY)
KEEP_AVG OF_FIELD(#SALES) IN_FIELD(#DIVAVG) BY_FIELD(#COMPANY)
KEEP_TOTAL OF_FIELD(#SALES) IN_FIELD(#COMTOT) BY_FIELD(#DIVNAM)
KEEP_AVG OF_FIELD(#SALES) IN_FIELD(#COMAVG) BY_FIELD(#DIVNAM)
KEEP_TOTAL OF_FIELD(#SALES) IN_FIELD(#GRDTOT)
KEEP_AVG OF_FIELD(#SALES) IN_FIELD(#GRDAVG)
PRINT LINE(#DETAIL)
ENDSELECT
ENDPRINT
```

Example 2: A sales file contains details of a company division (#DIVNAM) and of an entire year's sales in 4 quarters (#QTR01 -> #QTR04). Produce a report that summarizes total and average yearly sales by division.

```
DEF_BREAK NAME(#SUMMARY) FIELDS(#TOTAL #AVERAGE) TRIGGER
SELECT FIELDS(#DIVNAM #QTR01 #QTR02 #QTR03 #QTR04) FROM_FILE(SALEHIST)
KEEP_TOTAL OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(#DIVNAM)
KEEP_AVG OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(#DIVNAM)
PRINT LINE(*BREAKS)
ENDSELECT
ENDPRINT
```

7.63 KEEP_COUNT

The KEEP_COUNT command is used to keep count of field(s) changes in another field.

Note: Full RDMLX Fields cannot be used with this command.

The KEEP_COUNT command is only valid within SELECT or SELECTLIST command loops, because the processing logic used is implicitly linked to the SELECT / SELECTLIST loop logic.

Normally the KEEP_COUNT command is entered directly after the SELECT or SELECTLIST command. However there are specific exceptions to this. Refer to the examples section for more details.

Also See

[7.63.1 KEEP_COUNT Parameters](#)

[7.63.3 KEEP_COUNT Example](#)

[7.63.2 KEEP_COUNT Comments / Warnings](#)

Required

KEEP_COUNT --- OF_FIELD ---- list of field names -----
----->

| *expandable group expression* |
----- -- 50 max - -----

IN_FIELD ---- field name ----->

Optional

*BY_FIELD ---- *SELECTED -----|*
list of field names
| expandable group expression |
----- 20 max -----

7.63.1 KEEP_COUNT Parameters

BY_FIELD

IN_FIELD

OF_FIELD

OF_FIELD

Specifies from 1 to 50 fields that are to be counted by the KEEP_COUNT command.

Fields specified in this parameter must be defined in the LANSAs data dictionary or defined within this function by a DEFINE command. Expandable group expressions are valid in this parameter.

The fields specified may be numeric or non-numeric. It is possible to count the changes of a non-numeric field.

IN_FIELD

Specifies the field that is to hold the result of the count of the field(s) specified in the OF_FIELD parameter.

The field specified in this parameter must be numeric and must be defined in the LANSAs data dictionary or defined in this function with a DEFINE command.

This field is reset at the beginning of each SELECT/SELECTLIST loop.

BY_FIELD

Specifies the condition under which the fields are to be counted.

*SELECTED, which is the default value indicates, that the count should continue until the SELECT/SELECTLIST loop terminates (ie: all selected information).

Otherwise, specify a list of 1 to 20 field names (alternatively, enter an expandable group expression). The count continues until **one or more** of the fields in the list changes value. When one or more of the fields changes value the "accumulator" is reset to its null value and a new count cycle is started.

The "accumulator" is the work field keeping the count of the changes.

See the examples section for more details of how this parameter is used.

7.63.2 KEEP_COUNT Comments / Warnings

This applies to the commands KEEP_AVG, KEEP_COUNT, KEEP_MAX, KEEP_MIN and KEEP_TOTAL.

The processing logic used by these commands is best demonstrated by an example.

Consider the following SELECT loop that selects and prints invoice detail lines:

```
DEF_LINE NAME(#DETAIL) FIELDS(#INVNUM #VALUE)
DEF_BREAK NAME(#TOTAL) FIELDS(#INV_TOTAL) TRIGGER_BY(#I
DEF_BREAK NAME(#GRAND) FIELDS(#GRD_TOTAL)

SELECT FIELDS(#INVNUM #VALUE) FROM_FILE(INVLIN)
KEEP_TOTAL OF_FIELD(#VALUE) IN_FIELD(#INV_TOTAL)BY_FIELI
KEEP_TOTAL OF_FIELD(#VALUE) IN_FIELD(#GRD_TOTAL)
PRINT LINE(#DETAIL)
ENDSELECT

ENDPRINT
```

In highly simplified terms, what actually happens is:

```
Set #INV_TOTAL and #GRD_TOTAL to zero

---> Select next record

|   If #INVNUM has changed set #INV_TOTAL to zero
|   Add #VALUE to #INV_TOTAL
|   Add #VALUE to #GRD_TOTAL
|   If #INVNUM has changed, print "trailing" break
|     line #TOTAL
|   Print the #DETAIL line

---- Endselect
```

Print the grand total line #GRAND

Although this logic is highly simplified, it demonstrates the basic processing logic. This is shared by all KEEP_XXXXX commands and their relationship to the SELECT and SELECTLIST commands. Only the method of "accumulating" the result as an average, count, maximum, minimum or total varies.

Refer to the DEF_BREAK command for details of how and when the break lines are "triggered" and why the **correct** #INV_TOTAL value is always printed, even though it has apparently been cleared and reset by the KEEP_TOTAL command before it is printed.

If the KEEP_XXXXX command is within a condition (like IF or CASE/WHEN) and BY_FIELD is used the totals will not be reset correctly.

7.63.3 KEEP_COUNT Example

A personnel file contains details of employees with their department, section and salary. Print details of the count of departments and sections and the totals of salaries by department and section

```
DEF_LINE NAME(#DETAIL) FIELDS(#EMPNO #FULLNAME #DEPTME
*
DEF_BREAK NAME(#SECSUM) FIELDS(#DEPARTMENT #SECTION #DEM
DEF_BREAK NAME(#DEPSUM) FIELDS(#DEPARTMENT #DEM_DPTOT #
*
SELECT FIELDS(#DETAIL) FROM_FILE(PERSNEL)
KEEP_COUNT OF_FIELD(#EMPNO) IN_FIELD(#DEM_DPSEM) BY_FIE
KEEP_COUNT OF_FIELD(#EMPNO) IN_FIELD(#DEM_SCSEM) BY_FIE
KEEP_TOTAL OF_FIELD(#SALARY) IN_FIELD(#DEM_DPTOT) BY_FIE
KEEP_TOTAL OF_FIELD(#SALARY) IN_FIELD(#DEM_SCTOT) BY_FIE
PRINT LINE(#DETAIL)
ENDSELECT
*
ENDPRINT
```

7.64 KEEP_MAX

The KEEP_MAX command is used to keep the value of the field having the maximum value of field(s) in another field.

Note: Full RDMLX Fields cannot be used with this command.

The KEEP_MAX command is only valid within SELECT or SELECTLIST command loops, because the processing logic used is implicitly linked to the SELECT / SELECTLIST loop logic.

Normally the KEEP_MAX command is entered directly after the SELECT or SELECTLIST command. However there are specific exceptions to this. Refer to the examples section for more details.

Also See

[7.64.1 KEEP_MAX Parameters](#)

[7.64.3 KEEP_MAX Example](#)

[7.64.2 KEEP_MAX Comments / Warnings](#)

Required

*KEEP_MAX --- OF_FIELD ---- list of field names -----
----->*

*| expandable group expression |
----- -- 50 max - -----*

IN_FIELD ---- field name ----->

Optional

*BY_FIELD ---- *SELECTED -----|
list of field names
| expandable group expression |
----- 20 max -----*

7.64.1 KEEP_MAX Parameters

BY_FIELD

IN_FIELD

OF_FIELD

OF_FIELD

Specifies from 1 to 50 fields that are to be scanned for the maximum value by the KEEP_MAX command.

Fields specified in this parameter must be defined in the LANSAs data dictionary or defined within this function by a DEFINE command. Expandable group expressions are valid in this parameter.

The fields specified must also be numeric.

IN_FIELD

Specifies the field that is to hold the value of the field having the maximum value of the field(s) specified in the OF_FIELD parameter.

The field specified in this parameter must be numeric and must be defined in the LANSAs data dictionary or defined in this function with a DEFINE command.

This field is reset at the beginning of each SELECT/SELECTLIST loop.

BY_FIELD

Specifies the condition under which the fields are to be scanned for the maximum value.

*SELECTED, which is the default value indicates, that the scanning for maximum value should continue until the SELECT/SELECTLIST loop terminates (ie: all selected information).

Otherwise, specify a list of 1 to 20 field names (alternatively, enter an expandable group expression). The scanning for maximum value continues until **one or more** of the fields in the list changes value. When one or more of the fields changes value the "accumulator" is reset to its null value and a new scan for maximum value cycle is started.

The "accumulator" is the work field keeping track of the maximum value.

See the examples section for more details of how this parameter is used.

7.64.2 KEEP_MAX Comments / Warnings

Refer to the [7.63.2 KEEP_COUNT Comments / Warnings](#) which are also applicable to KEEP_MAX.

7.64.3 KEEP_MAX Example

A sales file contains details of a company division (#DIVNAM) and of an entire year's sales in 4 quarters (#QTR01 -> #QTR04). Produce a report that summarizes the maximum and minimum values and the total yearly sales by division.

```
DEF_BREAK  NAME(#SUMMARY) FIELDS(#TOTAL #MAXIMUM #MI
```

```
SELECT    FIELDS(#DIVNAM #QTR01 #QTR02 #QTR03 #QTR04) FROM  
KEEP_TOTAL OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD  
KEEP_MAX  OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(  
KEEP_MIN  OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(  
PRINT    LINE(*BREAKS)  
ENDSELECT
```

```
ENDPRINT
```

7.65 KEEP_MIN

The KEEP_MIN command is used to keep the value of the field having the minimum value of field(s) in another field.

Note: Full RDMLX Fields cannot be used with this command.

The KEEP_MIN command is only valid within SELECT or SELECTLIST command loops, because the processing logic used is implicitly linked to the SELECT / SELECTLIST loop logic.

Normally the KEEP_MIN command is entered directly after the SELECT or SELECTLIST command. However there are specific exceptions to this. Refer to the examples section for more details.

Also See

[7.65.1 KEEP_MIN Parameters](#)

[7.65.3 KEEP_MIN Example](#)

[7.65.2 KEEP_MIN Comments / Warnings](#)

Required

KEEP_MIN --- OF_FIELD ---- list of field names -----
---->

| expandable group expression |
----- -- 50 max - -----

IN_FIELD ---- field name ----->

Optional

*BY_FIELD ---- *SELECTED -----|*
list of field names
| expandable group expression |
----- 20 max -----

7.65.1 KEEP_MIN Parameters

BY_FIELD

IN_FIELD

OF_FIELD

OF_FIELD

Specifies from 1 to 50 fields that are to be scanned for the minimum value by the KEEP_MIN command.

Fields specified in this parameter must be defined in the LANSAs data dictionary or defined within this function by a DEFINE command. Expandable group expressions are valid in this parameter.

The fields specified must also be numeric.

IN_FIELD

Specifies the field that is to hold the value of the field having the minimum value of the field(s) specified in the OF_FIELD parameter.

The field specified in this parameter must be numeric and must be defined in the LANSAs data dictionary or defined in this function with a DEFINE command.

This field is reset at the beginning of each SELECT/SELECTLIST loop.

BY_FIELD

Specifies the condition under which the fields are to be scanned for the minimum value.

*SELECTED, which is the default value indicates, that the scanning for minimum value should continue until the SELECT/SELECTLIST loop terminates (ie: all selected information).

Otherwise, specify a list of 1 to 20 field names (alternatively, enter an expandable group expression). The scanning for minimum value continues until **one or more** of the fields in the list changes value. When one or more of the fields changes value the "accumulator" is reset to its null value and a new scan for minimum value cycle is started.

The "accumulator" is the work field keeping track of the minimum value.

7.65.2 KEEP_MIN Comments / Warnings

Refer to the [7.63.2 KEEP_COUNT Comments / Warnings](#) which are also applicable to KEEP_MIN.

7.65.3 KEEP_MIN Example

A sales file contains details of a company division (#DIVNAM) and of an entire year's sales in 4 quarters (#QTR01 -> #QTR04). Produce a report that summarizes the maximum and minimum values and the total yearly sales by division.

```
DEF_BREAK NAME(#SUMMARY) FIELDS(#TOTAL #MAXIMUM #MI
```

```
SELECT FIELDS(#DIVNAM #QTR01 #QTR02 #QTR03 #QTR04) FROM  
KEEP_TOTAL OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD  
KEEP_MAX OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(  
KEEP_MIN OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(  
PRINT LINE(*BREAKS)  
ENDSELECT
```

```
ENDPRINT
```

7.66 KEEP_TOTAL

The KEEP_TOTAL command is used to keep the total of field(s) in another field.

Note: Full RDMLX Fields cannot be used with this command.

The KEEP_TOTAL command is only valid within SELECT or SELECTLIST command loops, because the processing logic used is implicitly linked to the SELECT / SELECTLIST loop logic.

Normally the KEEP_TOTAL command is entered directly after the SELECT or SELECTLIST command. However there are specific exceptions to this. Refer to the examples section for more details.

Also See

[7.66.1 KEEP_TOTAL Parameters](#)

[7.66.3 KEEP_TOTAL Examples](#)

[7.66.2 KEEP_TOTAL Comments / Warnings](#)

Required

KEEP_TOTAL --- OF_FIELD ---- list of field names -----
----->

| expandable group expression |
----- -- 50 max - -----

IN_FIELD ---- field name ----->

Optional

*BY_FIELD ---- *SELECTED -----|*
list of field names
| expandable group expression |
----- 20 max -----

7.66.1 KEEP_TOTAL Parameters

BY_FIELD

IN_FIELD

OF_FIELD

OF_FIELD

Specifies from 1 to 50 fields that are to be totaled by the KEEP_TOTAL command.

Fields specified in this parameter must be defined in the LANSAs data dictionary or defined within this function by a DEFINE command. Expandable group expressions are valid in this parameter.

The fields specified must also be numeric.

IN_FIELD

Specifies the field that is to hold the result of the totaling of the field(s) specified in the OF_FIELD parameter.

The field specified in this parameter must be numeric and must be defined in the LANSAs data dictionary or defined in this function with a DEFINE command.

This field is reset at the beginning of each SELECT/SELECTLIST loop.

BY_FIELD

Specifies the condition under which the fields are to be totaled.

*SELECTED, which is the default value indicates, that the totaling should continue until the SELECT/SELECTLIST loop terminates (ie: all selected information).

Otherwise, specify a list of 1 to 20 field names (alternatively, enter an expandable group expression). The totaling continues until **one or more** of the fields in the list changes value. When one or more of the fields changes value the "accumulator" is reset to its null value and a new totaling cycle is started.

The "accumulator" is the work field calculating the total.

See the examples section for more details of how this parameter is used.

7.66.2 KEEP_TOTAL Comments / Warnings

Refer to the [7.63.2 KEEP_COUNT Comments / Warnings](#) which are also applicable to KEEP_TOTAL.

7.66.3 KEEP_TOTAL Examples

Example 1: A sales history file contains details of a company name (#COMPANY), a division name (#DIVNAM) and a sales value (#SALES). Print all sales details and print total and average sales details by division, company and report (grand total):

```
DEF_LINE NAME(#DETAIL) FIELDS(#COMPANY #DIVNAM #SALES)
DEF_BREAK NAME(#SUBDIV) FIELDS(#DIVTOT #DIVAVG) TRIGGER
DEF_BREAK NAME(#SUBCOM) FIELDS(#COMTOT #COMAVG) TRIGGER
DEF_BREAK NAME(#GRAND) FIELDS(#GRDTOT #GRDAVG)
SELECT FIELDS(#DETAIL) FROM_FILE(SALEHIST)
KEEP_TOTAL OF_FIELD(#SALES) IN_FIELD(#DIVTOT) BY_FIELD(#COMPANY)
KEEP_AVG OF_FIELD(#SALES) IN_FIELD(#DIVAVG) BY_FIELD(#COMPANY)
KEEP_TOTAL OF_FIELD(#SALES) IN_FIELD(#COMTOT) BY_FIELD(#COMPANY)
KEEP_AVG OF_FIELD(#SALES) IN_FIELD(#COMAVG) BY_FIELD(#COMPANY)
KEEP_TOTAL OF_FIELD(#SALES) IN_FIELD(#GRDTOT)
KEEP_AVG OF_FIELD(#SALES) IN_FIELD(#GRDAVG)
PRINT LINE(#DETAIL)
ENDSELECT
ENDPRINT
```

Example 2: A sales file contains details of a company division (#DIVNAM) and of an entire year's sales in 4 quarters (#QTR01 -> #QTR04). Produce a report that summarizes total and average yearly sales by division.

```
DEF_BREAK NAME(#SUMMARY) FIELDS(#TOTAL #AVERAGE) TRIGGER
SELECT FIELDS(#DIVNAM #QTR01 #QTR02 #QTR03 #QTR04) FROM_FILE(SALEHIST)
KEEP_TOTAL OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(#DIVNAM)
KEEP_AVG OF_FIELD(#QTR01 #QTR02 #QTR03 #QTR04) IN_FIELD(#DIVNAM)
PRINT LINE(*BREAKS)
ENDSELECT

ENDPRINT
```

7.67 LEAVE

The LEAVE command is a loop modifying command. It causes the current loop to be exited and execution to continue on the statement following the end loop command.

CONTINUE/LEAVE commands work inside all loop commands.

Also See

[7.67.1 LEAVE Parameters](#)

[7.67.2 LEAVE Comments / Warnings](#)

[7.67.3 LEAVE Examples](#)

[7.3 BEGIN_LOOP](#)

[7.15 CONTINUE](#)

[7.30 DUNTIL](#)

[7.31 DOWHILE](#)

[7.83 SELECT](#)

Required

LEAVE ----->

Optional

>-- *IF* ----- '*condition*' -----|

7.67.1 LEAVE Parameters

IF

IF

Optionally specifies the condition that is to be evaluated to determine if the LEAVE should be executed. If not specified the LEAVE is executed immediately. For more details, refer to [Specifying Conditions and Expressions](#).

7.67.2 LEAVE Comments / Warnings

The LEAVE loop modifying command operates as follows:

```
    < loop command >
---- LEAVE
|
|  < end loop command >
--->
```

The < loop command > and < end loop command > can be any of the following:

- SELECT / ENDSELECT
- SELECTLIST / ENDSELECT
- SELECT_SQL / ENDSELECT
- DOWHILE / ENDWHILE
- DUNTIL / ENDUNTIL
- BEGIN_LOOP / END_LOOP.

7.67.3 LEAVE Examples

[Using LEAVE within a SELECT Loop](#)

[Using LEAVE within a BEGIN_LOOP loop](#)

Using LEAVE within a SELECT Loop

At times, when one record has been read in a SELECT loop, there is no need to read further. This example demonstrates how to achieve this using the LEAVE command within a SELECT loop.

For example, assume you have a pricing file defined like this, sorted by these primary keys:

1. #ITEMNO (ascending).
2. #EFF_DATE (ascending).

Item No.	Effective Date	Price (\$)
(Packed 7,0)	(Packed 8,0)	(Packed 9,2)

116	2000/01/01	4.00
-----	------------	------

116	2000/07/01	4.25
-----	------------	------

116	2001/01/01	4.50
-----	------------	------

116	2001/07/01	4.75
-----	------------	------

116	2002/01/01	5.00
-----	------------	------

For the purpose of an invoice, a price for item 116 as at 08/02/2001 is required.

```
DEFINE FIELD(#REQITEM) REFFLD(#ITEMNO)
```

```
DEFINE FIELD(#REQDATE) REFFLD(#EFF_DATE)
```

```
DEFINE FIELD(#PRICEOUT) REFFLD(#PRICE)
```

```
BEGIN_LOOP
```

```
REQUEST FIELDS(#REQITEM #REQDATE)
```

```
CHANGE FIELD(#PRICEOUT) TO(0)
```

```
EXECUTE SUBROUTINE(GETPRICE) WITH_PARMs(#REQITEM #REQDATE)
```

```
DISPLAY FIELDS(#REQITEM #PRICEOUT)
```

```
END_LOOP
```

```

SUBROUTINE NAME(GETPRICE) PARMS((#REQITEM *RECEIVED) (#I
SELECT   FIELDS(#ITEMNO #EFF_DATE #PRICE) FROM_FILE(PRICIN
CHANGE   FIELD(#PRICEOUT) TO(#PRICE)
LEAVE
ENDSELECT
ENDROUTINE

```

The SELECT positions to the pricing record with an effective date the same as or earlier than the requested date and reads it. If a record is found, the value of #PRICE is moved into #PRICEOUT. As the required record has been found it is not necessary to keep reading pricing records, so the LEAVE is used to exit from the SELECT loop.

If no record is found for the requested item, prior to (or equal to) the requested date, the value #PRICEOUT remains zero.

Alternatively, instead of requesting an 'as of' date today's date can be used automatically.

```

DEFINE   FIELD(#REQITEM) REFFLD(#ITEMNO)
DEFINE   FIELD(#PRICEOUT) REFFLD(#PRICE)

```

```

BEGIN_LOOP
REQUEST  FIELDS(#REQITEM)
CHANGE   FIELD(#PRICEOUT) TO(0)
EXECUTE  SUBROUTINE(GETPRICE) WITH_PARMS(#REQITEM #PRI
DISPLAY  FIELDS(#REQITEM #PRICEOUT)
END_LOOP

```

```

SUBROUTINE NAME(GETPRICE) PARMS((#REQITEM *RECEIVED)
(#PRICEOUT *RETURNED))
SELECT   FIELDS(#ITEMNO #EFF_DATE #PRICE) FROM_FILE(PRICIN
CHANGE   FIELD(#PRICEOUT) TO(#PRICE)
LEAVE
ENDSELECT
ENDROUTINE

```

Using LEAVE within a BEGIN_LOOP loop

This example demonstrates how to use the LEAVE command within a

BEGIN_LOOP loop. With the help of an additional user function key and a defined condition the LEAVE command causes the loop to end when the user presses the 'Finish' key.

```
DEF_COND NAME(*FINISHED) COND('#IO$KEY = "09"')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #SURNAME #GIV

BEGIN_LOOP
REQUEST FIELDS(#EMPNO #SURNAME #GIVENAME) BROWSELIST
LEAVE IF(*FINISHED)
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
MESSAGE MSGTXT('Input of Employees Completed')
```

7.68 LOC_ENTRY

The LOC_ENTRY command is used to locate the first entry in a list that satisfies a specified condition.

The list specified must be a working list (used to store information within a program). It is not possible to use the LOC_ENTRY command against a browse list (used for displaying information at a workstation).

Refer to the DEF_LIST command for more details of lists and list processing.

Also See

[7.68.1 LOC_ENTRY Parameters](#)

[7.68.2 LOC_ENTRY Examples](#)

Optional

```
LOC_ENTRY ---- IN_LIST ----- *FIRST -----
--->
      list name

>-- WHERE ----- 'condition' ----->

>-- RET_NUMBER --- *NONE -----
-->
      field name

>-- RET_STATUS --- *STATUS -----
>
      field name

>-- RET_ENTRY ---- *YES -----|
      *NO
```

7.68.1 LOC_ENTRY Parameters

IN_LIST

RET_ENTRY

RET_NUMBER

RET_STATUS

WHERE

IN_LIST

Specifies the name of the list that should be searched by this command.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used. This list must have the TYPE(*WORKING) parameter to be valid.

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command and must have the TYPE(*WORKING) parameter.

WHERE

Refer to [Specifying Conditions and Expressions](#).

If no WHERE parameter is specified then the first entry in the list will always be returned by the LOC_ENTRY command.

RET_NUMBER

Optionally specify the name of a field that is to be set by the LOC_ENTRY command to contain the entry number of the entry located.

If no entry is found in the list that satisfies the condition the field's value is not changed by the LOC_ENTRY command.

*NONE, which is the default value indicates that no field is to contain the located list entry number. If a field is nominated, it must be of type numeric and have been defined previously in the LANSAs data dictionary or elsewhere in the function.

As each entry is added to a list by the ADD_ENTRY command it is assigned a number that identifies it. List entries are numbered from 1 (first entry number) to 9999 (maximum possible last entry number) sequentially.

By knowing a list entry number it is possible to directly retrieve an individual list entry (see the GET_ENTRY command for more details).

RET_STATUS

Specifies the name of a field that is to receive the "return code" that results from the LOC_ENTRY command.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2 and be defined in the LANSAs data dictionary or elsewhere in the function. Even if a user field is nominated the special field #IO\$STS is still updated.

The LOC_ENTRY command only returns 2 possible status codes. These are "OK" (an entry was located) or "NR" (no entry was located).

RET_ENTRY

Specifies whether or not the located list entry should be returned from the list into the RDML program.

*YES, which is the default value, indicates that the located list entry should be returned into the RDML program.

*NO indicates that there is no need to return the list entry into the RDML program. This option is typically used when a list is being used for validation purposes. In such cases, it is the fact that the entry is in (or not in) the list that matters, not its location or content.

The command:

```
LOC_ENTRY IN_LIST(#COUNTRIES) WHERE('#CNTRY = AUST')
```

is functionally identical to the commands:

```
LOC_ENTRY IN_LIST(#COUNTRIES) WHERE('#CNTRY = AUST') RET_
IF_STATUS IS(*OKAY)
GET_ENTRY NUMBER(#NUMBER) FROM_LIST(#COUNTRIES)
ENDIF
```

7.68.2 LOC_ENTRY Examples

Example 1: Locate the first entry in a list called #ORDERLINE where the product of quantity and price is greater than 1000:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA  
  
LOC_ENTRY IN_LIST(#ORDERLINE) WHERE('(#QUANTITY * #PRICE)
```

Example 2: A "table file" called #COUNTRY (a list of countries) contains fields #CCODE (country code), #CMNEM (country mnemonic) and #CNAME (country full name). Use a working list to create a subroutine that will minimize I/O's to the file (ie: replace FETCH commands with LOC_ENTRY commands):

```
SUBROUTINE NAME(GET_CNTRY) PARMS((#GETCODE *RECEIVED)  
(#CMNEM *RETURNED) (#CNAME *RETURNED))  
  
DEFINE FIELD(#GETCODE) REFFLD(#CCODE)  
DEF_LIST NAME(#COUNTRIES) FIELDS(#CCODE #CMNEM #CNAME  
  
LOC_ENTRY IN_LIST(#COUNTRIES) WHERE('#CCODE = #GETCODE')  
  
IF_STATUS IS_NOT(*OKAY)  
FETCH FIELDS(#COUNTRIES) FROM_FILE(COUNTRY) WITH_KEY(  
ADD_ENTRY TO_LIST(#COUNTRIES)  
ENDIF  
  
ENDROUTINE
```

Note that this routine makes no allowances for a country code not being found in the file #COUNTRY. The entry is added to the list regardless of whether the country was found or not.

Note also the use of field #GETCODE (instead of #CCODE) in the parameter list. If this was not done, the WHERE condition in the LOC_ENTRY command would have to be expressed as WHERE('#CCODE = #CCODE'), which is always true, so the first entry would always be retrieved.

When a field referenced in a WHERE condition is part of the working list, it is the occurrence of the field in the working list that is evaluated, not the actual field as it is known in the program.

Thus the condition WHERE('#CCODE = #GETCODE') is actually saying "where the value of field country code **in a working list entry** is equal to the value of field get code in the program".

7.69 MENU

The MENU command is used to cause an executing RDML program to end and a re-display of the process's main menu to occur. Note that the executing function **ends** and the process controller receives control again.

Using the MENU command is functionally identical to using the MENU function key.

Optionally a message may be issued which will be routed back onto the process controller's message queue. This message will appear on line 22/24 when the process's main menu is (re)displayed.

Note: The MENU command indicates control should return to the last displayed process menu. This means that in non-procedural or full RDMLX functions the MENU command cannot be sensibly or predicably used.

Also See

[7.69.1 MENU Parameters](#)

[7.69.2 MENU Examples](#)

Optional

```
MENU ----- MSGTXT -----*NONE -----
---->
           'message text'

>-- MSGID ----- *NONE ----->
           message identifier

>-- MSGF ----- *NONE ----->
           message file . library name

>-- MSGDTA ----- substitution variables -----|
           | expandable group expression |
           ----- 20 max -----
```

7.69.1 MENU Parameters

MSGDTA

MSGF

MSGID

MSGTXT

MSGTXT

Allows up to 80 characters of message text to be specified. This text will be displayed on line 22/24 when the process's main menu is (re)displayed. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID/MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be issued when the function ends and the process's main menu is (re)displayed. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the

field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAs aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA(''BOLTS  '' #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.69.2 MENU Examples

Example 1: End a function and re-display the process menu:

```
MENU MSGTXT('Previous function ended at user request')
```

Example 2: Cause a function to end and the process main menu to be re-displayed with message details from an IBM i message file:

```
MENU MSGID(USR0046) MSGF(QUSRMSG)
```

IBM i only:

```
MENU MSGID(USR0167) MSGF(QUSRMSG.*LIBL)
```

```
MENU MSGID(USR8046) MSGF(QUSRMSG.USERLIB01)
```

7.70 MESSAGE

The MESSAGE command has 4 main uses:

- To cause a message to appear on line 22/24 of the next screen format that is displayed to the user.
- To cause a message to be displayed on line 22/24 of the current screen format (no matter what it is) and then disappear when the next screen format is displayed to the user.
- To cause a message to be displayed on line 22/24 of the current screen format (no matter what it is) and then delay the executing job for a minimum period of time.
- To cause a single message to overlay the current screen format in a message "window" and optionally receive a reply to the message back into the RDML program. The message window may be positioned to overlay the top, middle or bottom of the current screen format.

Portability Considerations Refer to parameters: [MIN_TIME](#) and [TYPE](#).

Also See

[7.70.1 MESSAGE Parameters](#)

[7.70.2 MESSAGE Comments / Warnings](#)

[7.70.3 MESSAGE Examples](#)

[GET_MESSAGE Built In Function](#)

[CLR_MESSAGES Built In Function](#)

[MESSAGE_COLLECTOR Built In Function](#)

Optional

```
MESSAGE ----- MSGTXT -----*NONE -----
----->
           'message text'

>-- MSGID ----- *NONE ----->
           message identifier

>-- MSGF ----- *NONE ----->
           message file . library name
```

>-- *MSGDTA* ----- *substitution variables* ----->
|*expandable group expression*|
----- *20 max* -----

>-- *TYPE* ----- **INFO* ----->
**STATUS*
**WINDOW*
**WINDOWBUZ*

>-- *MIN_TIME* ----- *0* ----->
number of seconds

>-- *REPLY* ----- **NONE* ----->
field name

>-- *BORDER* ----- **YES*----->
**NO*

>-- *LOCATE* ----- **BOTTOM* -----|
**MIDDLE*
**TOP*

7.70.1 MESSAGE Parameters

BORDER

LOCATE

MIN_TIME

MSGDTA

MSGF

MSGID

MSGTXT

REPLY

TYPE

MSGTXT

Allows up to 80 characters of message text to be specified. This text will be displayed on line 22/24 of the current or next screen format that is displayed to the user. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID/MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be displayed on line 22/24 of the current or next screen displayed to the user. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

Portability Considerations **If you are relying on receiving specific message identifiers in your application, please note:**

On IBM i, when messages are routed between RDMLX components or functions and RDML functions, the original message identifier is not retained. Such message are re-issued as text messages with a message identifier of DCM9899.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA("BOLTS  " #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

TYPE

Specifies the type of message that is to be issued.

*INFO, which is the default value, causes the message to appear on line 22/24 of the NEXT screen format that is displayed to the user. This message will remain on line 22/24 until the user presses the enter key again.

*STATUS indicates that the message should appear on line 22/24 of the current screen, no matter what it is. The current screen may not even be part of the LANSAs system. When the next screen format is presented to the user the status message will be erased from the screen. Typically "status" messages are issued to inform the user that some extended action is in progress and that he/she should wait. See the following examples for more information.

*WINDOW indicates that the message should be presented to the user in a message "window". The window will overlay the current screen format (no matter what it is) and can be positioned at the top, middle or bottom of the screen (see the LOCATE parameter). Once the window has been presented the user must press the enter key before processing is resumed. When processing is resumed the screen is restored to what it was before the message window was displayed. Optionally a reply to the message may be received back into the RDML program (see the REPLY parameter).

*WINDOWBUZ indicates that the message should be presented to the user in a message "window". When it is presented the workstation alarm will sound. (See TYPE(*WINDOW) for more details.)

Portability *WINDOWBUZ is interpreted as *WINDOW and a build
Considerations warning is generated when used in Visual LANSAs.

MIN_TIME

Specifies the minimum time that a *STATUS message should be displayed on line 22/24 of the current screen. Use this parameter to extend the display time of a status message so that it can be read. If no value is specified for this parameter a default value of 0 seconds is assumed (which will cause the message to flash on the screen briefly). Otherwise specify the minimum display time required in seconds. Note that the user's job will wait (and not perform any useful work) for the number of seconds specified in this parameter.

This parameter can be used to "suspend" a function for a period of time while waiting for some other function to complete.

Portability Not supported in the current release of Visual LANSAs but
Considerations will be supported in a future release. A build warning will be generated when used in Visual LANSAs.

REPLY

Optionally nominates a field that is to receive the user's reply to the message. This parameter is only valid for messages using the TYPE(*WINDOW)

parameter.

*NONE, which is the default value, indicates that the user should not be given an option to reply to the message. In this case the message window will only display the message text to the user.

When a field name is specified it must be defined in the LANSAs data dictionary or in the function. The field may be alphanumeric or numeric. Where the field is alphanumeric and longer than 20 characters only the first 20 characters may be specified by the user. When the field is numeric it must be an integer (i.e. no decimal positions). When a field name is specified, the message window contains the message text and an input field identified by the string "Reply?" into which the user may enter the required reply to the message.

BORDER

Specifies for messages of TYPE(*WINDOW) whether or not the message window should be surrounded by a border. The default value is *YES. The only other possible value for this parameter is *NO, which indicates that no border is required.

LOCATE

Specifies for messages of TYPE(*WINDOW) where on the screen the message window should be located.

*BOTTOM, which is the default value, indicates that the window should be positioned at the bottom of the screen.

The other allowable values are *TOP and *MIDDLE, which indicate that the window should be positioned at the top or middle of the screen respectively.

7.70.2 MESSAGE Comments / Warnings

- MESSAGE TYPE(*WINDOW) or TYPE(*WINDOWBUZ) is not currently available for those applications that are enabled for the LANSA GUI (Graphical User Interface).
- MESSAGE TYPE(*WINDOW), TYPE(*WINDOWBUZ) or TYPE(*STATUS) are not available for those applications that are enabled for LANSA for the WEB.
- As many TYPE(*INFO) messages as required can be issued. The first message issued will be displayed on line 22/24 of the next screen presented to the user with a "+" sign indicating that more messages follow. These can be reviewed by using either the ROLL keys or the MESSAGE function key. For information, refer to [Messages and the Help Key](#) in the *LANSA for i User Guide*.
- Messages of TYPE(*WINDOW) should not be issued when a function is executing in batch.

7.70.3 MESSAGE Examples

[Issuing a Plain Text Message](#)

[Issuing Multiple Messages During an Abort](#)

[Issuing Messages with Dynamically Constructed Text](#)

[Issuing Messages with an *MTXT Variable as the Text](#)

[Issuing Messages with Substituted Variables](#)

[Using Messages to Instruct and Notify the User](#)

[Using Messages to Format Text](#)

Issuing a Plain Text Message

This example shows how to issue a text message that will be shown on the user's screen:

```
MESSAGE MSGTXT('Welcome to the LANSAsystem')
```

Issuing Multiple Messages During an Abort

This example shows how to issue multiple messages during an abort sequence:

```
MESSAGE MSGTXT('=====')
MESSAGE MSGTXT('== EMPLOYEE DETAILS NOT FOUND IN PSLM')
MESSAGE MSGTXT('=====')
MESSAGE MSGTXT('== FATAL ERROR - CONTACT YOUR SUPERVISOR')
ABORT MSGTXT('=====')
```

Issuing Messages with Dynamically Constructed Text

This subroutine demonstrates how to dynamically construct a message with all the relevant details at the time that it needs to be issued:

```
SUBROUTINE NAME(MESSAGE) PARMS((#MSGTXT1 *RECEIVED) (#MSGTXT2 *RECEIVED) (#MSGTXT3 *RECEIVED) (#MSGDTA *RECEIVED))
DEFINE FIELD(#MSGTXT1) TYPE(*CHAR) LENGTH(40) DECIMALS(0)
DEFINE FIELD(#MSGTXT2) REFFLD(#MSGTXT1)
DEFINE FIELD(#MSGTXT3) REFFLD(#MSGTXT1)
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132) DECIMALS(0)
USE BUILTIN(BCONCAT) WITH_ARGS(#MSGTXT1 #MSGTXT2 #MSGTXT3 #MSGDTA)
ABORT MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDROUTINE
```

The subroutine can then be used to issue messages like this:

```
EXECUTE SUBROUTINE(MESSAGE) WITH_PARMS('Details for employ
```

Or this:

```
EXECUTE SUBROUTINE(MESSAGE) WITH_PARMS(#DEPARTMENT ' dep
```

Issuing Messages with an *MTXT Variable as the Text

In multilingual applications you sometimes need to issue messages that contain *MTXT variables as their message text. This subroutine shows one way of doing this.

```
SUBROUTINE NAME(MTXTMESSGE) PARS((#MSGDTA *RECEIVED
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132) DECIMALS
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDROUTINE
```

The subroutine can then be used to send messages as follows;

```
EXECUTE SUBROUTINE(MTXTMESSGE) WITH_PARMS(*MTXTDEMI
```

Issuing Messages with Substituted Variables

This example shows how a message wording can be defined in a message file and the required details substituted as variables when the message is issued. The message would be defined like this:

Message File:	MYMSGF			
Message ID:	MSG0002			
Message Text:	'The salary for &1 &2 &3 is &4.'			
Substitution Variables				
Label	Type	Length	Dec.	Intended Content
&1	*CHAR	5		#EMPNO (Employee Number)
&2	*CHAR	20		#GIVENAME
&3	*CHAR	20		#SURNAME
&4	*DEC	11	2	#SALARY

Then the message is issued like this:

```
GROUP_BY  NAME(#XG_EMPLOY) FIELDS(#EMPNO #GIVENAME #S  
  
REQUEST  FIELDS(#EMPNO)  
FETCH    FIELDS(#XG_EMPLOY) FROM_FILE(PSLMST) WITH_KEY(#  
IF_STATUS IS(*OKAY)  
MESSAGE  MSGID(MSG0002) MSGF(MYMSGF) MSGDTA(#XG_EMPL  
ELSE  
MESSAGE  MSGTXT('Details for employee can not be found')  
ENDIF
```

Using Messages to Instruct and Notify the User

This example demonstrates how to use messages to both instruct the user on what they should do next and notify the user about what the program is currently doing. The example gives some input instructions then begins a cycle of requesting input, notifying the user of the various processing steps as it executes them and then finally notifies the user of the completion of processing before asking for the next input.

```
MESSAGE  MSGTXT('Input instructions appear here. Press Enter')  
  
BEGIN_LOOP  
REQUEST  FIELDS(#STD_TEXT)  
MESSAGE  MSGTXT('Processing Step 1. Please wait.') TYPE(*STATUS)  
EXECUTE  SUBROUTINE(WAIT)  
MESSAGE  MSGTXT('Processing Step 2. Please wait.') TYPE(*STATUS)  
EXECUTE  SUBROUTINE(WAIT)  
MESSAGE  MSGTXT('Processing Step 3. Please wait.') TYPE(*STATUS)  
EXECUTE  SUBROUTINE(WAIT)  
MESSAGE  MSGTXT('Processing has completed. Please input next')  
END_LOOP  
  
SUBROUTINE NAME(WAIT)  
BEGIN_LOOP TO(20000000)  
END_LOOP  
ENDROUTINE
```

If the TYPE parameter of the *STATUS messages was changed to *INFO then the messages would not appear until after the processing had been completed and the next REQUEST command is executed. The messages would then remain until the next REQUEST command is executed.

Using Messages to Format Text

This example demonstrates how messages can be used to format variables and text into concatenated text without the need to convert data types or concatenate strings. The following messages definitions are created:

Message File:	MYMSGF			
Message ID:	MSG0004			
Message Text:	'The name of employee &1 is &2 &3.'			
Substitution Variables				
Label	Type	Length	Dec.	Intended Content
&1	*CHAR	5		#EMPNO (Employee Number)
&2	*CHAR	20		#GIVENAME
&3	*CHAR	20		#SURNAME

Message File:	MYMSGF			
Message ID:	MSG0005			
Message Text:	'The department and section of employee &1 is &2 &3.'			
Substitution Variables				
Label	Type	Length	Dec.	Intended Content
&1	*CHAR	5		#EMPNO (Employee Number)
&2	*CHAR	20		#DEPARTMENT
&3	*CHAR	20		#SECTION

Message File:	MYMSGF			
Message ID:	MSG0006			
Message Text:	'The salary of employee &1 is &2.'			
Substitution Variables				
Label	Type	Length	Dec.	Intended Content
&1	*CHAR	5		#EMPNO (Employee Number)
&2	*DEC	11	2	#SALARY

Then, these messages are issued with the appropriate variables. The GET_MESSAGES BIF is used in the subroutine to copy the messages to a browse list to be displayed on the screen and, if the user wants, to lines of a report for printing:

```

DEFINE FIELD(#PRINT) TYPE(*CHAR) LENGTH(1) LABEL('PRINT?')
DEFINE FIELD(#EMPTXT) TYPE(*CHAR) LENGTH(78)
DEFINE FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPTXT)
DEF_LINE NAME(#EMPLOYEE) FIELDS(#EMPTXT)

```

```

REQUEST FIELDS(#PRINT)
CLR_LIST NAMED(#EMPBROWSE)

```

```

SELECT FIELDS(*ALL) FROM_FILE(PSLMST1)
MESSAGE MSGID(MSG0004) MSGF(MYMSGF) MSGDTA(#EMPNO #C
MESSAGE MSGID(MSG0005) MSGF(MYMSGF) MSGDTA(#EMPNO #I
MESSAGE MSGID(MSG0006) MSGF(MYMSGF) MSGDTA(#EMPNO #S
IF COND('#TERMDATE *NE 0')
MESSAGE MSGID(MSG0003) MSGF(MYMSGF) MSGDTA(#EMPNO #C
ENDIF
ENDSELECT

```

```

EXECUTE SUBROUTINE(SHOWMSG)

```

```
DISPLAY BROWSELIST(#EMPBROWSE)
```

```
SUBROUTINE NAME(SHOWMSG)  
USE BUILTIN(GET_MESSAGE) TO_GET(#RETCODE #EMPTXT)  
DOWHILE COND('#RETCODE = OK')  
ADD_ENTRY TO_LIST(#EMPBROWSE)  
IF COND('#PRINT *NE N')  
PRINT LINE(#EMPLOYEE)  
ENDIF  
USE BUILTIN(GET_MESSAGE) TO_GET(#RETCODE #EMPTXT)  
ENDWHILE  
ENDROUTINE
```

The wording of the messages in the message file can be changed or even translated into another language without the need to change or recompile the program.

7.71 ON_ERROR

The ON_ERROR command is used to transfer control to another command if an "error condition" is raised within a validation block.

For more information about the raising of the "error condition" inside a validation block refer to the ENDCHECK command.

An optional error message may be issued when the "error condition" is raised.

Also See

[7.71.1 ON_ERROR Parameters](#)

[7.71.2 ON_ERROR Comments / Warnings](#)

[7.71.3 ON_ERROR Examples](#)

Required

```
ON_ERROR ----- GOTO ----- label -----  
--->
```

```
>-- MSGTXT ----- *NONE ----->  
      message text
```

```
>-- MSGID ----- *NONE ----->  
      message identifier
```

```
>-- MSGF ----- DC@M01 . *LIBL -----  
>  
      message file . library name
```

```
>-- MSGDTA ----- substitution variables -----|  
      |expandable group expression |  
      ----- 20 max -----
```

7.71.1 ON_ERROR Parameters

GOTO

MSGDTA

MSGF

MSGID

MSGTXT

GOTO

Specifies the label associated with the command to which control will be passed if an "error condition" has been raised within a validation block.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type

*CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA('"BOLTS  "' #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.71.2 ON_ERROR Comments / Warnings

The ON_ERROR command can only be used within a BEGINCHECK / ENDCHECK validation block. Refer to these commands for more details of validation blocks.

7.71.3 ON_ERROR Examples

The following example applies to the ON_ERROR command.

Consider the following RDML "input and validate" program:

```
REQUEST  FIELDS(#A #B #C #D #E)

BEGINCHECK
FILECHECK FIELD(#A) USING_FILE(ACHECK)
FILECHECK FIELD(#B) USING_FILE(BCHECK)
FILECHECK FIELD(#C) USING_FILE(CCHECK)
FILECHECK FIELD(#D) USING_FILE(DCHECK)
FILECHECK FIELD(#E) USING_FILE(ECHECK)
ENDCHECK
```

This validation block is relatively inefficient because it performs all subsequent checks even if a previous check failed. For instance if field #A is in error then the checks of #B, #C, #D and #E are wasted because they will have to be performed again when field #A is corrected by the user.

The ON_ERROR command can be used to improve the efficiency of such a validation block by specifying a series of "premature ends" like this:

```
L10: REQUEST  FIELDS(#A #B #C #D #E)

        BEGINCHECK
        FILECHECK FIELD(#A) USING_FILE(ACHECK)
        ON_ERROR  GOTO(L10)
        FILECHECK FIELD(#B) USING_FILE(BCHECK)
        ON_ERROR  GOTO(L10)
        FILECHECK FIELD(#C) USING_FILE(CCHECK)
        ON_ERROR  GOTO(L10)
        FILECHECK FIELD(#D) USING_FILE(DCHECK)
        ON_ERROR  GOTO(L10)
        FILECHECK FIELD(#E) USING_FILE(ECHECK)
        ENDCHECK
```

Thus any failed check will cause control to be passed back to the REQUEST command and the error details will be displayed for correction. Of course the disadvantage of this technique is that if a field is in error all subsequent fields

will not be validated until the first error is corrected.

7.72 OPEN

The OPEN command is used to open (or control the opening of) the **database** file(s) specified by the FILE parameter. Individual files or all files can be opened or controlled.

There are two forms of the OPEN command.

The first is the **executable** form. It is called executable because the file is opened at the time the OPEN command is executed. This form supports access to the IBM i operating system command OPNQRYF (Open Query File).

The second is the **declarative** form. It is called declarative because the OPEN command does not actually execute. The presence of the command in the RDML program declares how and when the file is to be automatically opened by LANSAs.

These 2 forms of the command are described in more detail later.

There is normally no need to code OPEN or CLOSE commands into RDML programs. LANSAs will automatically open and close files as required. OPEN and CLOSE commands are only used to specifically open or close a file, or to modify how and when LANSAs automatically opens and closes them.

Portability Considerations FILE (library) is not supported in Visual LANSAs. A build warning will be generated if used in Visual LANSAs code. Refer to parameters: [ALWCPYDTA](#), [OPTIMIZE](#), [SEQONLY](#), [COMMIT](#) and [TYPE](#) , [ALWCPYDTA](#), [OPTIMIZE](#), [SEQONLY](#), [COMMIT](#) and [TYPE](#) , [FILE](#) , [KEYFLD](#), [ALWCPYDTA](#), [OPTIMIZE](#), [SEQONLY](#), [COMMIT](#) and [TYPE](#) , [QRYSLT](#), [ALWCPYDTA](#), [OPTIMIZE](#), [SEQONLY](#), [COMMIT](#) and [TYPE](#) , [ALWCPYDTA](#), [OPTIMIZE](#), [SEQONLY](#), [COMMIT](#) and [TYPE](#) and [USE_OPTION](#) .

Also See

[7.72.1 OPEN Parameters](#)

[7.72.2 OPEN Comments / Warnings](#)

[7.72.3 OPEN Examples](#)

Optional

`OPEN ----- FILE----- *ALL. *FIRST -----`
->

file name.library name

```
>-- USE_OPTION --- *FIRSTSCREEN -----  
----->  
      *IMMEDIATE  
      *ONDEMAND  
      *OPNQRYP  
      *KEEPOPEN  
  
>-- IO_STATUS ---- *STATUS ----->  
      field name  
  
>-- IO_ERROR ----- *ABORT -----  
>  
      *NEXT  
      *RETURN  
      label  
  
>-- QRYSLT ----- *ALL ----->  
      'selection criteria'  
      #field name  
      '=EXCHANGE'  
  
>-- KEYFLD ----- *NONE ----->  
      *FILE  
      'key / sort order required'  
      #field name  
  
>-- ALWCPYDTA ---- *YES ----->  
      *NO  
  
>-- OPTIMIZE ----- *ALLIO ----->  
      *FIRSTIO  
      *MINWAIT  
  
>-- SEQONLY ----- *YES --- number of records ----  
->  
      *NO
```

>-- *COMMIT* ----- **NO* ----->
 **YES*

>-- *TYPE* ----- **NORMAL* -----|
 **PERM*

7.72.1 OPEN Parameters

ALWCPYDTA, OPTIMIZE, SEQONLY, COMMIT and TYPE
ALWCPYDTA, OPTIMIZE, SEQONLY, COMMIT and TYPE

FILE

IO_ERROR

IO_STATUS

KEYFLD

ALWCPYDTA, OPTIMIZE, SEQONLY, COMMIT and TYPE
QRYSLT

ALWCPYDTA, OPTIMIZE, SEQONLY, COMMIT and TYPE

ALWCPYDTA, OPTIMIZE, SEQONLY, COMMIT and TYPE

USE_OPTION

FILE

Specifies the file(s) to be opened or controlled. Individual files can be specified or the default of *ALL can be nominated. For more information, refer to [Specifying File Names in I/O commands](#).

Note: The combination of parameters FILE(*ALL) and USE_OPTION(*OPNQRYF) is invalid. To open a query file, a specific file must be nominated in the FILE parameter.

Portability FILE (library) is not supported in Visual LANSA. A build
Considerations warning will be generated if used in Visual LANSA code.

USE_OPTION

Specifies the open or control option that LANSA is to use when opening this file. The allowable values for this parameter are:

*FIRSTSCREEN(FSS), which is the default value and is a "declarative" form of the command. This value indicates that the nominated file open(s) should be "overlapped" with the first screen interaction with the user. This means that the first screen is presented to the user, and while he/she is keying in data, the file(s) are opened.

*IMMEDIATE(IMD), which is an "executable" form of the command. This value indicates that the nominated file(s) should be opened immediately (ie: when the OPEN command is executed).

*ONDEMAND(OND), which is a "declarative" form of the command. This value specifies that the nominated file(s) should be opened as required (ie: on a demand basis). This means that the first I/O request to the file will cause it to be automatically opened. If no I/O request is ever made to the file it will not be opened.

*OPNQRYP(OUQ), which is an "executable" form of the command. This value indicates that the IBM i operating system command OPNQRYP should be used during the file open to select and/or order the data in the file. The file is opened when the OPEN command is executed.

OPNQRYP is a very powerful and useful command. However, it can cause significant performance degradation in some situations. Refer to the appropriate IBM supplied manuals for more details of the OPNQRYP command, its associated parameter values and performance impact.

*KEEPOPEN(KPO), which is an "executable" form of the command. This value indicates that the file(s) should be opened immediately (i.e., when the OPEN command is executed) and then left open until such time as they are closed by issuing a specific CLOSE command. Once a file has been opened this way it can only be closed from within LANSAs by issuing a specific CLOSE command.

The automatic close logic used by LANSAs when an RDML program terminates, cannot close a file that has been opened with this option.

This option is normally used for performance reasons to ensure that a frequently used file is left open at all times.

- Portability** *FIRSTSCREEN ignored with no known effect to the application.
- Considerations**
- *ONDEMAND ignored with no known effect to the application.
 - *OPNQRYP is only supported for execution on IBM i. On all other platforms an execution error will occur, but execution of the code can be made conditional.
 - *KEEPOPEN ignored. Testing of application required. A build warning will be generated when used in Visual LANSAs.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

For values refer to *I/O Return Codes*.

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command. The purpose of *NEXT is to permit you to handle error messages in the RDML, and then ABORT, rather than use the default ABORT. (It is possible for processing to continue for LANSAs on IBM i and Visual LANSAs, but this is NOT a recommended way to use LANSAs.)

ER returned from a database operation is a fatal error and LANSAs does not expect processing to continue. The IO Module is reset and further IO will be as if no previous IO on that file had occurred. Thus you must not make any presumptions as to the state of the file. For example, the last record read will not be set. A special case of an IO_ERROR is when a trigger function is coded to return ER in TRIG_RETC. The above description applies to this case as well. Therefore, LANSAs recommends that you do NOT use a return code of ER from a trigger function to cause anything but an ABORT or EXIT to occur before any further IO is performed.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

QRYSLT

Portability Not supported in the current release of Visual LANSA unless
Considerations using LANSA/SuperServer to an IBM i server.

Use of QRYSLT('=EXCHANGE') only supports RDML fields. Note that when fields are put on the exchange list in RDMLX, trailing blanks are stripped from the value put on the exchange list.

This parameter is only valid when used with the USE_OPTION(*OPNQRYF) parameter. It is ignored in all other cases. It is used to specify the selection criteria that should be used by the OPNQRYF command when building an access path to the data in the file.

It may be specified as an alphanumeric literal like this:

```
OPEN FILE(STATES) USE_OPTION(*OPNQRYF) QRYSLT('STATE *EQ "I
```

or as the name of a field that contains the selection criteria, like this:

```
CHANGE FIELD(#SELECTION) TO('STATE *EQ "NSW")  
OPEN FILE(STATES) USE_OPTION(*OPNQRYF) QRYSLT(#SELECTION
```

or, making use of the exchange list, like this:

```
OPEN FILE(STATES) USE_OPTION(*OPNQRYF) QRYSLT('=EXCHANGE
```

The second version indicates that the RDML program can change the selection criteria at execution time. By modifying the content of field #SELECTION the actual data selected by the program can be modified. This is one of the powerful facilities available with the OPNQRYF command. Note that Visual LANSA has similar behavior to this with the SELECT_SQL command, although it is only available with RDMLX objects on IBM i. Following is an example of cross-platform code. Great care must be taken to construct the query in a cross-platform manner, in particular, single quotes must be used around literals and the file must be closed on IBM i:

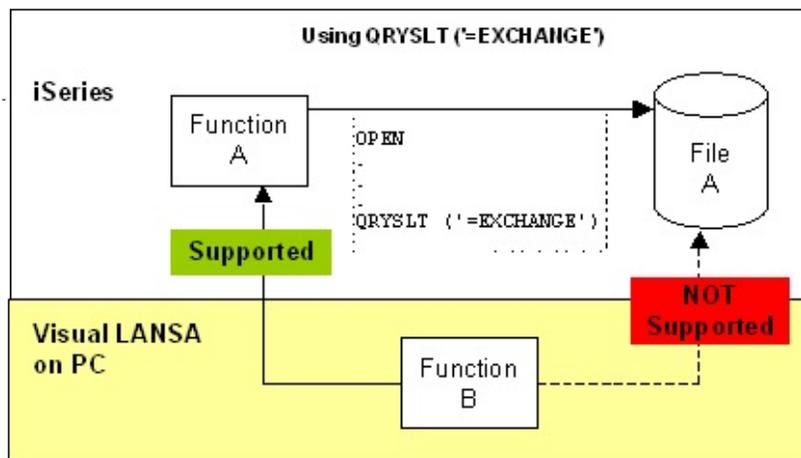
```
EXECUTE SUBROUTINE(QUOTE) WITH_PARM('NSW' #QUOTED)  
USE BUILTIN(BCONCAT) WITH_ARGS('STATE =' #QUOTED) TO_GET(#  
IF COND(*CPUTYPE *NE AS400)  
SELECT_SQL FIELDS(#STATE) FROM_FILES((#STATES)) WHERE(#SI  
DISPLAY FIELDS(#STATE)  
ENDSELECT  
ELSE
```

```

OPEN FILE(STATES) USE_OPTION(*OPNQRYF) QRYSLT(#SELECTIO
SELECT FIELDS(#STATE) FROM_FILE((#STATES))
  DISPLAY FIELDS(#STATE)
ENDSELECT
CLOSE FILE(STATES)
ENDIF
SUBROUTINE NAME(QUOTE) PARMS((#TEXT1 *RECEIVED) (#TEXT2
USE BUILTIN(TCONCAT) WITH_ARGS(*QUOTE #TEXT1 *QUOTE) TO
ENDROUTINE

```

The third version allows you to specify up to 256 characters in length per field and as many exchange fields as required to define a query select. '=EXCHANGE' is used in conjunction with the EXCHANGE command. This would be a better option than the second version if the query select is complicated and requires more than 256 characters to define. **This third version is only supported when executed locally on an IBM i. If you want to use it from Visual LANSA, an IBM i function MUST be executed via LANSA/SuperServer as shown in this example:**



Note: The use of the " (double quote) symbols within the QRYSLT parameter. The " (double quote) symbol can be used interchangeably with the ' (single quote) symbol by OPNQRYF. When using OPNQRYF through LANSA the " symbol is preferable because it is easier to code into alphanumeric literals. The field name STATE is not preceded by a # (hash) symbol in this example. This is because the actual selection request is being made to the OPNQRYF command, not to LANSA.

Refer to the OPNQRYF command in the appropriate IBM supplied manuals for

more details of the QRYSLT parameter and the values, operations and options that it supports.

KEYFLD

This parameter is only valid when used with the USE_OPTION(*OPNQRYF) parameter. It is ignored in all other cases.

It is used to specify the key fields that should be used by the OPNQRYF command when building an access path to the data in the file. This then allows access to the data in the file in the requested key order. It may be specified as an alphanumeric literal like this:

```
OPEN FILE(STATES) USE_OPTION(*OPNQRYF)
      KEYFLD('STATE POSTCD')
```

or as the name of a field that contains the key field names, like this:

```
CHANGE FIELD(#ORDER_BY) TO('STATE POSTCD')
OPEN FILE(STATES) USE_OPTION(*OPNQRYF) KEYFLD(#ORDER_BY)
```

The second version indicates that the key fields (and thus the order of processing data from the file) can be changed by the program at execution time. By modifying the content of field #ORDER_BY the order records are processed from file STATES, can be dynamically modified. This is another of the powerful facilities available with the OPNQRYF command.

Also note that the field names STATE and POSTCD are not preceded by a # (hash) symbol in this example. This is because the actual key/order request is being made to the OPNQRYF command, not to LANSAs.

Refer to the OPNQRYF command in the appropriate IBM supplied manuals for more details of the KEYFLD parameter and the values, operations and options that it supports.

Portability Not supported in the current release of Visual LANSAs unless
Considerations using LANSAs/SuperServer to an IBM i server.

**ALWCPYDTA,
OPTIMIZE,
SEQONLY,
COMMIT and
TYPE**

These parameters are only valid when used with the USE_OPTION(*OPNQRYP) parameter. They are ignored in all other cases. Refer to the OPNQRYP command in the appropriate IBM supplied manuals for more details of these parameters and the values, operations and options that they support.

Portability These parameters are not supported in the current release of
Considerations Visual LANSA unless using LANSA/SuperServer to an IBM i Server.

7.72.2 OPEN Comments / Warnings

- When no specific OPEN command is coded for a file in an **interactive** RDML program, the file is opened as if an OPEN USE_OPTION(*FIRSTSCREEN) command was used.
- When no specific OPEN command is coded for a file in a **batch** RDML program, the file is opened as if an OPEN USE_OPTION(*ONDEMAND) command was used.
- Any OPEN command or request will be ignored if the file is already open. The exception to this rule is when the *OPNQRYF option is used. In this case the open file is closed, and then re-opened with the appropriate OPNQRYF parameters.

In addition, use of the *KEEPOPEN option against a file that is already open will cause the file to be flagged so that it can only be closed by a specific CLOSE command. The file is not actually closed and opened again (as with the *OPNQRYF option), but it is internally tagged so that it cannot be closed unless a specific CLOSE request command is issued.

There are 2 ways a file can be closed:

- The first is by letting the normal RDML program termination logic CLOSE the file automatically.
- The second is to issue a specific CLOSE command in the RDML program.

When an RDML program terminates normally it will attempt to automatically close all the database files it uses. However, if one or more of the following conditions apply the close request for a file will be ignored:

- the file has already been closed or has never been opened.
- the associated process has the usage HEAVY option.
- the file has been previously opened with the *KEEPOPEN option.
- the function attempting to close the file is not the one that originally opened the file.
- A specific CLOSE command (ie: coded into the RDML program) will only be ignored if the file is already closed. In all other situations it will force the file to be closed, even if it was opened with the *KEEPOPEN option, or by some other program.
- If any file is referenced by an RDML command before it is specifically opened (ie: by an OPEN command) or before the appropriate automatic open

time has occurred (ie: a first screen interaction) it will be automatically opened on an "on demand" basis. What this means is that attempting any I/O operation against a closed file will merely result in the file being automatically opened.

- When using USE_OPTION(*OPNQRYF) it is advisable to code a specific CLOSE into the RDML program. This will ensure that any temporary access path created by the OPNQRYF command is destroyed.

In addition, after using the *OPNQRYF option in combination with a specific KEYFLD parameter (other than the special *FILE parameter), it is advisable to only use a simple SELECT loop to read all records from the temporary access path created by the OPNQRYF command.

Attempting to use a SELECT command (or any other I/O command for that matter) that has a WITH_KEY parameter may produce unpredictable results because the file key being used by the I/O module does not match the actual key of the temporary file created by the OPNQRYF command.

- When using the USE_OPTION(*FIRSTSCREEN) parameter the open "overlap" is attempted when the first screen is presented to the user. This screen is the first DISPLAY, REQUEST or POP_UP command that is actually executed in the program (which may not be the first one coded in the program).

Additionally, use of this option may cause a slight degradation over communication lines because it is not possible to use the DFRWRT (defer screen write until read) option in the associated IBM i display file.

Refer to the IBM i CRTDSPF command in the appropriate IBM supplied manual for more details of the DFRWRT parameter.

Attempting to manually change the display file to DFRWRT(*YES) will completely nullify the "overlap" feature and the user will have to wait until the file(s) are open before the first screen appears.

7.72.3 OPEN Examples

Example 1: Open the customer master file CUSTMST during the first screen interaction.

```
OPEN FILE(CUSTMST)
```

or

```
OPEN FILE(CUSTMST) USE_OPTION(*FIRSTSCREEN)
```

Example 2: Open all files immediately, except for the customer master file which should be opened "on demand".

```
OPEN FILE(*ALL) USE_OPTION(*IMMEDIATE)  
OPEN FILE(CUSTMST) USE_OPTION(*ONDEMAND)
```

Example 3: Open file STATES, load all records into working list #WRK_LIST and then close file STATES.

```
OPEN FILE(STATES) USE_OPTION(*IMMEDIATE)  
SELECT FIELDS(#WRK_LIST) FROM_FILE(STATES)  
ADD_ENTRY TO_LIST(#WRK_LIST)  
ENDSELECT  
CLOSE FILE(STATES)
```

Example 4: Open file STATES, load all records into working list #WRK_LIST in state name (STATNM) order, and then close file STATES.

```
OPEN FILE(STATES) USE_OPTION(*OPNQRYF) KEYFLD('STATNM')  
SELECT FIELDS(#WRK_LIST) FROM_FILE(STATES)  
ADD_ENTRY TO_LIST(#WRK_LIST)  
ENDSELECT  
CLOSE FILE(STATES)
```

Example 5: Open file STATES, load all records into working list #WRK_LIST in state name (STATNM) order, select only records where the post code (POSTCD) is in the range 2000 to 3999, and then close file STATES.

```
OPEN FILE(STATES) USE_OPTION(*OPNQRYF) KEYFLD('STATNM')  
SELECT FIELDS(#WRK_LIST) FROM_FILE(STATES)  
ADD_ENTRY TO_LIST(#WRK_LIST)
```

```
ENDSELECT
CLOSE FILE(STATES)
```

Example 6: Create a completely user driven name and address report that will answer just about any name and address query request. The name and address file is called NAMES.

```
DEFINE FIELD(#SELECTION) TYPE(*CHAR) LENGTH(100) LABEL
DEFINE FIELD(#ORDER_BY) TYPE(*CHAR) LENGTH(100) LABEL
DEF_LINE NAME(#LINE1) FIELDS(#CUSTNO #NAME #ADDR1 #AD
L1: REQUEST FIELDS(#SELECTION #ORDER_BY)

OPEN FILE(NAMES) USE_OPTION(*OPNQRYF) IO_ERROR(L1) QI

SELECT FIELDS(#LINE1) FROM_FILE(NAMES)
PRINT LINE(#LINE1)
ENDSELECT

CLOSE FILE(NAMES)
```

Example 7: Create a report that will allow the user to scan the name and address file for a suburb name in any of the 3 address lines. Print details of all customers that contain the required suburb.

```
DEFINE FIELD(#SRCHSUB) TYPE(*CHAR) LENGTH(20) LABEL('Subu
DEFINE FIELD(#SELECT) TYPE(*CHAR) LENGTH(256)
DEF_LINE NAME(#LINE1) FIELDS(#CUSTNO #NAME #ADDR1 #ADDI
REQUEST FIELDS(#SRCHSUB)
USE BUILTIN(TCONCAT) WITH_ARGS(''(ADDR1 *CT '''' #SRCHSUB ''
USE BUILTIN(TCONCAT) WITH_ARGS(#SELECT '' *OR (ADDR2 *CT ''
USE BUILTIN(TCONCAT) WITH_ARGS(#SELECT '' *OR (ADDR3 *CT ''
OPEN FILE(NAMES) USE_OPTION(*OPNQRYF) QRYSLT(#SELECT)
SELECT FIELDS(#LINE1) FROM_FILE(NAMES)
PRINT LINE(#LINE1)
ENDSELECT
CLOSE FILE(NAMES)
```

Example 8: Open file STATES, load all records into browse list #BROWSEL, select only records where the states = 'NSW' and 'QLD' and the post code is within the range of 2000 and 4999.

```
DEFINE FIELD(#FIELD A) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#FIELD B) TYPE(*CHAR) LENGTH(256)
CHANGE FIELD(#FIELD A) TO('(DEPARTMENT *EQ "ADM") *OR (DEPT
CHANGE FIELD(#FIELD B) TO('(DEPARTMENT *EQ "FLT") *OR (DEPTM
EXCHANGE FIELDS(#FIELD A)
EXCHANGE FIELDS(#FIELD B)
OPEN FILE(STATES) USE_OPTION(*OPNQRYF) QRYSLT('=EXCHAN
SELECT FIELDS((#BROWSEL)) FROM_FILE(STATES)
ADD_ENTRY TO_LIST(#BROWSEL)
ENDSELECT
```

7.73 OTHERWISE

The OTHERWISE command is used within a CASE / ENDCASE block in conjunction with WHEN commands.

The OTHERWISE command is used to indicate what command(s) should be executed if none of the WHEN commands are matched and executed.

Refer to the CASE, WHEN and ENDCASE commands for more details and examples of these commands.

Inclusion of an OTHERWISE command within a CASE / ENDCASE block is optional. However if one is used it should follow the last WHEN command and precede the ENDCASE command.

Also See

[7.73.1 OTHERWISE Parameters](#)

[7.73.2 OTHERWISE Examples](#)

[7.8 CASE](#)

[7.34 ENDCASE](#)

[7.100 WHEN](#)

OTHERWISE ----- no parameters -----
--|

7.73.1 OTHERWISE Parameters

No parameters exist for the OTHERWISE command.

7.73.2 OTHERWISE Examples

Refer to the [7.8 CASE](#) for use of OTHERWISE command

7.74 OVERRIDE

The **OVERRIDE** command is used to override the data dictionary attributes of a field in a function or component.

Any field attribute (except the type) can be overridden.

Any overrides specified take effect only within the function. The LANSAs data dictionary remains totally unaffected by the override.

Also See

[7.74.1 OVERRIDE Parameters](#)

[7.74.2 OVERRIDE Examples](#)

Required

```
OVERRIDE ----- FIELD ----- field name -----  
-->
```

Optional

```
>-- LENGTH ----- *SAME ----->  
    numeric value  
    incr/decr  *PLUS  
              *MINUS  
              *NONE  
    # to incr/decr *NONE  
    numeric value
```

```
>-- DECIMALS ----- *SAME -----  
>  
    numeric value  
    incr/decr  *PLUS  
              *MINUS  
              *NONE  
    # to incr/decr *NONE  
    numeric value
```

```
>-- LABEL ----- *SAME ----->  
    label name
```

```

>-- DESC ----- *SAME ----->
      text description

>-- COLHDG ----- *SAME -----
>
      column heading
      |         |
      -- 3 maximum --

>-- EDIT_CODE ---- *SAME -----
>
      edit code

>-- EDIT_WORD ---- *SAME -----
>
      edit word

>-- INPUT_ATR ---- *SAME -----
>
      input attributes

->-- OUTPUT_ATR --- *SAME -----
      output attributes

>-- DEFAULT ----- *SAME ----->
      default value

>-- TO_OVERLAY --- *NONE ----- 1 -----
à
      #field name  start position

>-- SHIFT ----- *SAME -----|
      keyboard shift

```

7.74.1 OVERRIDE Parameters

COLHDG

DECIMALS

DEFAULT

DESC

EDIT_CODE

EDIT_WORD

FIELD

INPUT_ATR

LABEL

LENGTH

OUTPUT_ATR

SHIFT

TO_OVERLAY

FIELD

Specifies the name of the field which is to be overridden. The field name must start with a # and be defined in the LANSAs data dictionary.

LENGTH

Specifies the length to which the field is to be overridden. If the value *SAME is specified the length of the field is not to be overridden. For specific information on allowable field lengths see [Field Types](#)

Type	Notes for Length parameter
*DEC or synonym *PACKED	If the field is an RDML field, changing the length to 31 or higher will make the working field an RDMLX field.
*SIGNED	If the field is an RDML field, changing the length to 31 or higher will make the working field an RDMLX field.
*DATE	Dates are fixed size (always 10) incr/decr must be *NONE # to incr/decr must be *NONE

*TIME	Times are fixed size (always 8) incr/decr must be *NONE # to incr/decr must be *NONE
*DATETIME	19, 21-29. The various lengths influence the number of fractional seconds. This must be made clear. A length of 19 means no fractional seconds, 21 - 29 means 1 - 9 fractional seconds. The DECIMALS parameter has no impact. incr/decr must be *NONE # to incr/decr must be *NONE

"Incr/decr" value is used in conjunction with *SAME on the length parameter. The purpose of this field is to allow the length to be incremented or decremented by a specific amount. Permissible values are *PLUS, *MINUS and *NONE.

- *PLUS specifies that the field length attribute is to be increased.
- *MINUS specifies that the field length attribute is to be decreased.
- *NONE specifies that the field length attribute is to remain the same.

"# to incr/decr" value is used in conjunction with the *SAME value on the length parameter and is directly related to the "incr/decr" value. The purpose of this field is to specify the value by which the field length value is to be increased or decreased. Permissible values for this field are a numeric value or the value *NONE.

Notes:

Until release 4PC E2, LANSA supported up to 15-digit long numeric fields. PC E2 introduced support for 30-digit long numeric fields. Please note that when you use this command to override the length of fields which are 15 digit or less with a length of greater than 15, specify *DBOPTIMISE in the OPTIONS parameter of the FUNCTION command to ensure backward compatibility.

Similarly, do not override the length of a field that is now at least 15 digits long with a length of 15 digits or less without specifying *DBOPTIMISE. This restriction arises from the fact that when support for longer numeric fields was

introduced, existing I/O modules could not handle conversion of 8-byte fields to 16-byte fields or vice versa.

DECIMALS

Specifies the number of decimal positions to which the field is to be overridden. If the value *SAME is specified then the decimal positions are not to be overridden. Otherwise a value in the range 0 to 63 must be specified.

"Incr/decr" value is used in conjunction with *SAME on the decimals parameter. The purpose of this field is to allow the decimal positions value to be incremented or decremented by a set amount. Permissible values are *PLUS, *MINUS and *NONE. *PLUS specifies that the field decimal positions attribute is to be increased. *MINUS specifies that the field decimal positions attribute is to be decreased. *NONE specifies that the field decimal positions attribute is to remain the same.

"# to incr/decr" value is used in conjunction with the *SAME value on the decimal positions parameter and is directly related to the "incr/decr" value. The purpose of this field is to specify the value by which the field decimal positions value is to be increased or decreased. Permissible values for this field are a numeric value or the value *NONE.

LABEL

Specifies the 15 character label which should be assigned to this field. *SAME indicates the label is not to be overridden. In all other cases specify the override label in quotes.

DESC

Specifies the 40 character description that should be assigned to this field. *SAME indicates the field description is not to be overridden. In all other cases specify the override description in quotes.

COLHDG

Specifies the 3 x 20 character column headings that should be assigned to this field. *SAME indicates the column headings are not to be overridden. In all other cases specify the 3 override column headings required in quotes.

EDIT_CODE

Specifies the field edit code for numeric fields. *SAME indicates the field edit code is not to be overridden. In other cases specify one of the edit codes from the list below as the override edit code:

Fields of type Integer, Signed, or Packed may have an Editcode or Editword, or may leave both as *SAME. However, Integer does not allow edit codes W and Y. All other field types must have EDIT_CODE(*SAME) EDIT_WORD(*SAME).

Edit codes supported by LANSAs are shown in [Standard Field Edit Codes](#).

EDIT_WORD

Specifies the override edit word to be assigned to the field. *SAME indicates that the edit word is not to be overridden. In other cases specify the override edit word as required.

Use of edit words should only be attempted by experienced users as the validity checking done by LANSAs is unsophisticated.

Fields of type Integer, Signed, or Packed may have an Editcode or Editword, or may leave both as *SAME. All other field types must have EDIT_CODE(*SAME) EDIT_WORD(*SAME).

Note that when overriding a field length and using EDIT_WORD(*SAME) you are specifying that the edit word associated with the data dictionary should be used. However, if the length or number of decimal positions used are different to the data dictionary definition the associated edit word may be invalid. In such cases it will be necessary to override the edit word as well.

Note also that the operating system handles edit words involving floating currency symbols on screen panels differently to how they are handled on reports. In such cases, it is suggested that a separate field (or a "virtual" field) is used for report production.

When an edit word is defined in LANSAs via the RDML command language it should be enclosed in triple quotes as opposed to single quotes.

For example:

Correct method for overriding an edit word for a 5,2 numeric field requiring a trailing %:

```
OVERRIDE FIELD(#INCREASE) TYPE(#DEC) LENGTH(5) DECIMALS(2)
      LABEL('Sales Increase') EDIT_WORD(' . %')
```

Incorrect method for overriding an edit word for a 5,2 numeric field requiring a trailing %:

```
OVERRIDE FIELD(#INCREASE) TYPE(#DEC) LENGTH(5) DECIMALS(2)
      LABEL('Sales Increase') EDIT_WORD(' . %')
```

Refer to IBM manual *Data Description Specifications* for more details. See keyword EDTWRD.

INPUT_ATR

Specifies the input attribute overrides that are required. *SAME indicates that no override of input attributes is required.

For information on allowable attributes for RDMLX fields see [Field Types](#).

Valid input attributes for types A (alphanumeric), P (packed), and S (signed) are:

Attribute	Description / Comments	A	P	S
AB	Allow to be blank.	Y	Y	Y
ME	Mandatory entry check required.	Y	Y	Y
MF	Mandatory fill check required.	Y	Y	Y
M10	Modulus 10 check required.		Y	Y
M11	Modulus 11 check required.		Y	Y
VN	Valid name check required.	Y		
FE	Field exit key required.	Y	Y	Y
LC	Lowercase entry allowed. If you do NOT set this attribute, refer to <i>PC Locale uppercasing requested</i> in Review or Change a Partition's Multilingual Attributes in the <i>LANSA for i User Guide</i> .	Y		
RB	Right adjust and blank fill.		Y	Y
RZ	Right adjust and zero fill.		Y	Y
RL	Move cursor right to left.	Y	Y	Y
RLTB	Tab cursor right/left top/bottom. Valid in SAA/CUA partitions only. Affects all screen panels in function.	Y	Y	Y
GRN	Display with color green.	Y	Y	Y
WHT	Display with color white.	Y	Y	Y
RED	Display with color red.	Y	Y	Y

TRQ	Display with color turquoise.	Y	Y	Y
YLW	Display with color yellow.	Y	Y	Y
PNK	Display with color pink.	Y	Y	Y
BLU	Display with color blue.	Y	Y	Y
BL	Display blinking.	Y	Y	Y
CS	Display with column separators.	Y	Y	Y
HI	Display in high intensity.	Y	Y	Y
ND	Non-display (hidden field).	Y	Y	Y
RA	Auto record advance field	Y	Y	Y
SREV	Store in reversed format. This special attribute is provided for bi-directional languages is not applicable in this context.	Y	N	N
SBIN	Store in binary format. This special attribute is provided for repository fields & is not applicable in this context.	Y	N	N
HIND	HINDI Numerics. Display using HINDI numerals. Refer to Hindi Numerics in the <i>LANSA for i User Guide</i> .	N	Y	Y
CBOX *	Check Box	Y	N	N
RBnn *	Radio Button	Y	N	N
PBnn *	Push Button	Y	N	N
DDxx *	Drop Down	Y	N	N

Attributes marked with an * represent the field with the corresponding GUI WIMP construct. Refer to [GUI WIMP Constructs](#) in the *LANSA for i User Guide* for more information

In partitions that comply with **SAA/CUA guidelines** the following attributes may be used as well (and are in fact preferred to those described above):

Attribute	Description / Comments
ABCH	Action bar and pull-down choices
PBPT	Panel title
PBPI	Panel identifier
PBIN	Instructions to user
PBFP	Field prompt / label / description details
PBBR	Brackets
PBCM	Field column headings
PBGH	Group headings
PBNT	Normal text
PBET	Emphasized text
PBEN *	Input capable field (normal)
PBEE *	Input capable field (emphasized)
PBCH	Choices shown on menu
PBSC	Choice last selected from menu
PBUC	Choices that are not available
PBCN	Protected field (normal)
PBCE	Protected field (emphasized)
PBSI	Scrolling information
PBSL	Separator line
PBWB	Pop-up window border
FKCH	Function key information

Note: Normally only PBEN and PBEE would be specified as input attributes. Refer to *SAA/CUA Implementation* in the *LANSAs Application Design Guide*

more details of these attributes. Also note that only one color can be specified for a field. Use of colors may affect other attributes. Refer to IBM manual *Data Description Specifications* for more details. Keywords that should be reviewed are CHECK, COLOR and DSPATR.

OUTPUT_ATR

Specifies the output attribute overrides that are required. *SAME indicates that no output attribute overrides are required. In other cases specify the required output attribute overrides from the list below:

For information on allowable attributes for RDMLX fields see [Field Types](#)

Valid output attributes for types Alpha (A), Packed (P), and Signed (S) are:

Attribute	Description / Comments	A	P	S
GRN	Display with color green.	Y	Y	Y
WHT	Display with color white.	Y	Y	Y
RED	Display with color red.	Y	Y	Y
TRQ	Display with color turquoise.	Y	Y	Y
YLW	Display with color yellow.	Y	Y	Y
PNK	Display with color pink.	Y	Y	Y
BLU	Display with color blue.	Y	Y	Y
BL	Display blinking.	Y	Y	Y
CS	Display with column separators.	Y	Y	Y
HI	Display in high intensity.	Y	Y	Y
ND	Non-display (hidden field).	Y	Y	Y
SREV	Store in reversed format. This special attribute is provided for bi-directional languages is not applicable in this context.	Y	N	N
SBIN	Store in binary format. This special attribute is provided for repository fields & is not applicable in this context.	Y	N	N
Urxx	User Defined Reporting Attribute. Provides access to IBM i DDS statements for printer files.	Y	Y	Y

	Refer to User Defined Reporting Attributes in the <i>LANSA for i User Guide</i> .			
HIND	HINDI Numerics. Display using HINDI numerals. Refer to Hindi Numerics in the <i>LANSA for i User Guide</i> .	N	Y	Y
CBOX *	Check Box	Y	N	N
RBnn *	Radio Button	Y	N	N
PBnn *	Push Button	Y	N	N
DDxx *	Drop Down	Y	N	N

Attributes marked with an * represent the field with the corresponding GUI WIMP construct. Refer to [GUI WIMP Constructs](#) in the *LANSA for i User Guide* for more information

In partitions that comply with **SAA/CUA guidelines** the following attributes may be used as well (and are in fact preferred to those described above):

Attribute	Description / comments
ABCH	Action bar and pull-down choices
PBPT	Panel title
PBPI	Panel identifier
PBIN	Instructions to user
PBFP	Field prompt / label / description details
PBBR	Brackets
PBCM	Field column headings
PBGH	Group headings
PBNT	Normal text
PBET	Emphasized text

PBEN	Input capable field (normal)
PBEE	Input capable field (emphasized)
PBCH	Choices shown on menu
PBSC	Choice last selected from menu
PBUC	Choices that are not available
PBCN *	Protected field (normal)
PBCE *	Protected field (emphasized)
PBSI	Scrolling information
PBSL	Separator line
PBWB	Pop-up window border
FKCH	Function key information

Note: Normally only PBCN and PBCE would be specified as output attributes. Refer to *SAA/CUA Implementation* in the *LANSA Application Design Guide* for more details of these attributes. Also note that only one color can be specified for a field. Use of colors may affect other attributes. Refer to IBM manual *Data Description Specifications* for more details. Keywords that should be reviewed are COLOR and DSPATR.

DEFAULT

Specifies the default value which is to apply to the field. *SAME indicates that no override of the field's default value is required. In other cases specify the override default value that is to apply to the field.

For information on what DEFAULT(*DEFAULT) means for RDMLX fields see [Field Types](#)

Default values specified can be:

- A system variable such as *BLANKS, *ZERO, *DATE or any other specifically defined at your installation.
- An alphanumeric literal such as BALMAIN.
- A numeric literal such as 1, 10.43, -1.341217.

- A process parameter such as *UP01.
- Another field name such as #ORDNUM.

TO_OVERLAY

Specifies that the field being overridden is to fully or partially overlay (ie: occupy the same storage locations) as the field referenced in this parameter.

It is invalid for RDMLX fields to be overlaid or overlay another field.

*NONE, which is the default value, indicates that the field is to occupy its own storage area and not to overlay any other field.

The only other allowable value that can be specified here is the name of another field defined in this program or the data dictionary, optionally followed by a starting position.

The TO_OVERLAY parameter is a powerful facility that allows a field to occupy the same storage (ie: memory locations) as another field. The power of this parameter means that you must understand exactly what it causes to happen and what problems you may cause yourself in using it.

The following notes and comments should be read in full before attempting to use this parameter:

- You must **NOT** overlay a field onto a field that is itself overlaid onto another field. This is NOT checked by the full function checker and may cause a compile failure.
- You should fully understand the IBM i data storage formats of **character**, **signed/zoned decimal** and **packed decimal** before attempting to overlay fields of varying types. Overlaying of fields means that you can easily cause invalid decimal data to be placed into decimal fields, thus causing your program to fail in an unpredictable manner.
- The start position component of this parameter allows you to overlay just a part of a specific field, rather than its entire length. The start position is a full byte position, even when using packed decimal fields. When you specify a start position you **MUST** ensure that you do not overlay the field beyond the end position of the field being overlaid.

This is **NOT** checked by the full function checker. Failure to observe this rule can cause dangerous and unpredictable results.

- Array index fields must not be overlaid on or by other fields (in any context).
- A packed decimal field of even length can be overlaid on another field,

however the RPG compiler will always interpret the overlaying field as the next highest odd length. For example, if the data dictionary contains 2 packed decimal (type P) fields of length 6,0 called #DEC6 and #OVR6 then the following will cause #DEC6 to be treated by the RPG compiler as a packed decimal (6,0) value:

```
    OVERRIDE FIELD(#OVR6) TO_OVERLAY(#DEC6)
```

The #OVR6 will be treated by the RPG compiler as a packed decimal (7,0) value. There is no memory length problem here, both fields require 4 bytes of memory to be stored, it is just the way that the RPG compiler works that may cause a presentation length problem on reports.

- When the data validation commands RANGECHECK, VALUECHECK, DATECHECK, CALLCHECK, CONDCHECK, FILECHECK or SET_ERROR are used on an overlaying field they also set an error for the overlaid field. For example, if the data dictionary contains fields #INPUT (character length 3), #INPC1 (character length 1) and #INPC3 (character length 1), then this code accepts a 3 character field (#INPUT) from the workstation and validates that the first character is an A, B or C and also that the last character is an X, Y or Z:

```
    OVERRIDE FIELD(#INPC1) TO_OVERLAY(#INPUT 1)
    OVERRIDE FIELD(#INPC3) TO_OVERLAY(#INPUT 3)
```

```
    REQUEST FIELDS(#INPUT)
```

```
    BEGINCHECK
    VALUECHECK FIELD(#INPC1) WITH_LIST('A' 'B' 'C')
    VALUECHECK FIELD(#INPC3) WITH_LIST('X' 'Y' 'Z')
    ENDCHECK
```

When an error is triggered against overlaid fields #INPC1 or #INPC3 by the VALUECHECK commands, it is also triggered against the overlaid field #INPUT. This means that when the REQUEST command is (re)executed in an error situation, field #INPUT will be displayed in reverse video.

- You should consider the following points, if you are continually overriding fields into overlay positions in **every** program such as in this example:

OVERRIDE FIELD(#COMPANY) TO_OVERLAY(#ACCOUNT 1)
OVERRIDE FIELD(#DEPTMENT) TO_OVERLAY(#ACCOUNT 4)
OVERRIDE FIELD(#SECTION) TO_OVERLAY(#ACCOUNT 6)
OVERRIDE FIELD(#SUBACC) TO_OVERLAY(#ACCOUNT 7)

- The data model and/or function model behind your entire design may be slightly suspect. Clearly, there are 4 separate "elements": a company, a department, a section and a "subaccount" number. Why are they being aggregated into a field called "account"? Is this really necessary? Is it just being done because this is the way it was done before? What are the alternatives? Have they been fully considered and investigated? Do they offer a new and easier to understand system perspective to end users of the system?
- If the structure is to be implemented, then possibly the overlay logic should be moved to the "virtual field" area, thus centralizing the logic and saving programmers from having to repeat and maintain the logic in every program.

SHIFT

Specifies the keyboard shift to be used to override any existing keyboard shift. If no keyboard shift is specified then the value *SAME is assumed.

For information on what values of SHIFT, apart from *DEFAULT, are valid for each working field type see [Field Types](#).

For working fields of type Boolean, SHIFT must be *DEFAULT.

*SAME indicates that the keyboard shift is not to be overridden.

Refer to the IBM manual *Data Description Specifications* for more details.

Position 35 for display files is the entry that should be reviewed.

7.74.2 OVERRIDE Examples

Example 1: Override attributes of a field called #ORDER because the context in which it is used makes it appear like a "last" order number:

```
OVERRIDE FIELD(#ORDER) DESC('Last Order Number') LABEL('Last Or
```

Example 2: Override field #QTY to have 3 more significant digits

```
OVERRIDE FIELD(#QTY) LENGTH(*SAME *PLUS 3)
```

Example 3: Override field #QTY so that it has 3 more significant digits and 2 more decimal digits:

```
OVERRIDE FIELD(#QTY) LENGTH(*SAME *PLUS 5) DECIMALS(*SAM
```

Example 4: Override the length of field #SHORT to 10 characters

```
OVERRIDE FIELD(#SHORT) LENGTH(10)
```

Example 5: Override field #SHORT so that it is 10 characters shorter:

```
OVERRIDE FIELD(#SHORT) LENGTH(*SAME *MINUS 10)
```

Example 6: Override a numeric field that is keyboard shift S so that it can be displayed with the J edit code

```
OVERRIDE FIELD(#SHIFTS) EDIT_CODE(J) SHIFT(Y)
```

7.75 POINT

Except when using SELECT_SQL, the POINT command is used to "point", "re-direct" or "override" all I/O requests made to a file to:

- Another file in the same library
- Another file in another library
- The same file in another library
- A specific member within a file.

This command is provided for backward compatibility rather than for its specific functionality. See the [Permanent File Overrides](#) in the *LANSA for i User Guide* for a generally better mechanism for achieving similar functionality.

Portability Considerations The POINT command is supported in RDMLX code for compatibility with existing RDML code. When executed on platforms other than IBM i it has no effect.

Code generation varies for RDML functions and RDMLX code, and a difference may be caused in the library that is used where there are multiple files of the same name. RDML function generation on IBM i matches up File references to Libraries using the Library List of the job that is compiling the object; if not found in the library list, the first File in the repository found in EBCDIC collation sequence will be used. RDMLX objects are generated on Windows, which does not have a Library List, and so the first file in the repository found in ANSI collation sequence order will be used. Thus if the POINT command specifies a library but an IO command does not (or vice versa), and there are multiple files of the same name, the RPG and C code generation may match different libraries.

Also See

[7.75.1 POINT Parameters](#)

[7.75.2 POINT Comments/Warnings](#)

[7.75.3 POINT Examples](#)

Required

POINT ----- *FILE* ----- *file name* . **FIRST* -----
>
library name

Optional

>-- *TO_FILE* - ---- **SAME* ----->
file name

>-- *TO_LIBRARY* --- **LIBL* ----->
library name

>-- *TO_MEMBER* ---- **FIRST* -----
|
member name

7.75.1 POINT Parameters

FILE

TO_FILE

TO_LIBRARY

TO_MEMBER

FILE

For details of how file names are specified, refer to *Specifying File Names in I/O commands*. The name specified here is the name of the file as it **is known** or has **been coded** within this function.

TO_FILE

Specifies the name of the file to which all I/O requests are to be actually directed when the function executes. If the default value *SAME is used the name of the file is not changed from its current value.

The file name can be specified as an alphanumeric literal (e.g.: CUSTMST), or an alphanumeric variable name (e.g.: #FILE) or an alphanumeric system variable (e.g.: *PERIODFILE).

TO_LIBRARY

Specifies the name of the library in which the file can be found. If default value *LIBL is used the library list of the function will be searched at execution time to locate the file. If it cannot be found an error will occur.

The library name can be specified as an alphanumeric literal (e.g.: QGPL), an alphanumeric variable name (e.g.: #USINGLIB) or an alphanumeric system variable (e.g.: *COMPANYLIB).

TO_MEMBER

Specifies the name of the member within the file that is to be used. If the default value *FIRST is used then the first member within the file will be used.

Refer to the appropriate IBM manuals for more details of file members and multi-member files.

Use of multi-member files within new applications is not recommended because of the maintenance problems involved. This facility is provided primarily as a means of accessing existing databases that contain multi-member files.

The member name can be specified as an alphanumeric literal (e.g.: COMP010), an alphanumeric variable name (e.g.: #USINGMBR) or an alphanumeric system

variable (e.g.: *COMPANYMBR).

7.75.2 POINT Comments/Warnings

- When an RDML program is **first invoked** (or subsequently invoked in LIGHT usage mode) the internal details of all files, libraries and members to be used are initialized as if the command is executed for every file used by the program.

```
POINT FILE(XXXXX) TO_FILE(XXXXX) TO_LIBRARY(*LIBL)
      TO_MEMBER(*FIRST)
```

This means that by default files are opened with the same name as they were coded, using the library list to locate the file and accessing the first member in the file.

- The POINT command is an **executable** command. The internal details of which file, library and member are to be used are updated when the POINT command executes.
- Interactive RDML functions attempt to **overlap database file** opens with the first screen interaction. So it is advisable to execute all required POINT commands before issuing the first DISPLAY, REQUEST or POP_UP command.
- File re-direction details provided in a POINT command apply only to the **current function**, not to called or subsequently invoked functions. They must specify their own POINT commands.
- When a POINT command is issued against a **logical file** you must also specify a POINT command for the associated/underlying physical file. This is because many I/O requests made via the logical file are actually performed via the physical file (e.g.: UPDATE, DELETE, LOCK(*YES)). This requirement is checked by the full function checker which will issue a fatal error if it is not complied with.
- For similar reasons, when a POINT command is issued against a physical file, POINT commands must also be issued for any associated logical files that are **used** by the function. This requirement is checked by the full function checker which will issue a fatal error message if it is not complied with.
- It is possible using POINT commands that a physical file is re-directed to a particular member and an associated logical view is re-directed to another member. In this situation strange and unpredictable results may occur. For

instance a record read from member A via a logical view and then updated, may actually update a completely different record in member B.

- **Take care when using the POINT command.** Make sure that the logical and physical file POINT commands involved ultimately "point" to data in the same physical file member.
- When the actual file, library or member being accessed is to be changed while a function is executing the file must be closed via a CLOSE command or the I/O module will issue a fatal error.

For instance, this command will fail on the second FETCH:

```
POINT FILE(CUSTMST) TO_MEMBER(CURRENT)
FETCH FIELDS(#NAME) FROM_FILE(CUSTMST) WITH_KEY(#CUS
POINT FILE(CUSTMST) TO_MEMBER(ARCHIVE)
FETCH FIELDS(#NAME) FROM_FILE(CUSTMST) WITH_KEY(#CUS
```

The I/O module will abort with an error indicating that member CURRENT is open and you have asked for an I/O to member ARCHIVE.

The correct way to achieve this is as follows:

```
POINT FILE(CUSTMST) TO_MEMBER(CURRENT)
FETCH FIELDS(#NAME) FROM_FILE(CUSTMST) WITH_KEY(#CUS
CLOSE FILE(CUSTMST)
POINT FILE(CUSTMST) TO_MEMBER(ARCHIVE)
FETCH FIELDS(#NAME) FROM_FILE(CUSTMST) WITH_KEY(#CUS
```

- Where entire application systems use multi-member files there is usually a predictable naming convention used for file members. For example, in multi-company financial systems the member name may be an "M" followed by the company number. In such cases it is worth creating system variables that automatically set the name of the member to be used.

Consider the following examples of standard coding at the beginning of all RDML programs used in a multi-member financial system:

```
POINT FILE(GLMAST) TO_MEMBER(*COMPANY_MBR)
POINT FILE(SUMAST) TO_MEMBER(*COMPANY_MBR)
```

POINT FILE(FTMAST) TO_MEMBER(*COMPANY_MBR)

The same feature can be used to determine the name of the library that is to be used. For instance a system that utilizes 3 identical data libraries called PRODUCTION, TESTING1 and TESTING2 might use the following coding at the beginning of all RDML programs:

```
POINT FILE(GLMAST) TO_LIBRARY(*DATA_LIBRARY)
POINT FILE(SUMAST) TO_LIBRARY(*DATA_LIBRARY)
POINT FILE(FTMAST) TO_LIBRARY(*DATA_LIBRARY)
```

The program behind the system variable "*DATA_LIBRARY" works out the required library name from the user profile or some other identifier.

- In the **current release** of LANSAs, files used by I/O modules for other purposes such as data validation and batch control logic cannot be re-directed. They are always opened by searching the library list and using the first member in the file.
- Where I/O is re-directed to a file that has **not been defined** to LANSAs it is not possible for LANSAs to check the user's access rights against its internal security information. In such cases the user is granted "special access" and a warning message is issued. Such access is still subject to normal IBM i operating system security.
- It is the developer's **responsibility** to check and test the ramifications of using the POINT command in any RDML program.

7.75.3 POINT Examples

Example 1: Point file CUSTMST to library TESTDATA:

```
POINT FILE(CUSTMST) TO_LIBRARY(TESTDATA)
```

or

```
DEFINE FIELD(#LIBRARY) TYPE(*CHAR) LENGTH(10)  
CHANGE FIELD(#LIBRARY) TO('TESTDATA')  
POINT FILE(CUSTMST) TO_LIBRARY(#LIBRARY)
```

Example 2: Point file GLTRANS to a member with the same name as the current workstation:

```
POINT FILE(GLTRANS) TO_MEMBER(*JOB)
```

or

```
DEFINE FIELD(#MEMBER) TYPE(*DEC) LENGTH(2) DECIMALS(0)  
CHANGE FIELD(#MEMBER) TO(*JOB)  
POINT FILE(GLTRANS) TO_MEMBER(#MEMBER)
```

Example 3: Point file GLTRANS to a member with a name made up of an "M" concatenated with the current job number:

```
DEFINE FIELD(#MEMBER) TYPE(*CHAR) LENGTH(10)  
USE BUILTIN(CONCAT) WITH_ARGS('M' *JOBNBR) TO_GET(#MEMBER)  
POINT FILE(GLTRANS) TO_MEMBER(#MEMBER)
```

7.76 POP_UP

The POP_UP command displays information on a workstation in a pop up window.

The POP_UP command is only valid in RDMLX functions when being used on the Web. If it is used elsewhere a fatal error occurs at runtime. If this occurs, either put your POP_UP command in an RDML function or use a Form to show user information.

The POP_UP command is functionally very similar to the DISPLAY command, except that the information is presented in a window that overlays (or pops up over) information that is already on the screen.

For example, the following DISPLAY command:

```
DISPLAY FIELDS(#CUSTNUM #NAME #ADDR1 #ADDR2 #ADDR3 #PH
```

might cause a panel to be presented on the workstation that looks something like this:

```
Customer no : 99999999
Name       : XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Address    : XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
           : XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
           : XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Telephone  : 9999999999
Zip        : 999999
```

If the following POP_UP command was then executed:

```
POP_UP FIELDS(#DEBIT #CREDIT)
```

the resulting panel on the workstation might look something like this:

```
Customer no : 99999999
Name       : XXXXXXXX ----- XX
Address    : XXXXXXXX |           | XX
           : XXXXXXXX | Debit : 999999.99 | XX
           : XXXXXXXX | Credit : 999999.99 | XX
```

Telephone : 99999999 | |
Zip : 999999 -----

Portability Considerations Refer to parameters: [FIELDS](#) , [IGCCNV_KEY](#) and [TEXT](#) .

Also See

- [7.76.1 POP_UP Parameters](#)
- [7.76.2 POP_UP Comments / Warnings](#)
- [7.76.3 POP_UP Examples](#)

Optional

```
POP_UP ----- FIELDS -----  
- field name field attributes --->  
      |           |           ||  
      |           --- 7 max ----- |  
      | expandable group expression |  
      |----- 1000 max for RDMLX----|  
      |----- 100 max for RDML ----|  
  
>-- DESIGN ----- *ACROSS ----->  
      *DOWN  
  
>-- IDENTIFY ----- *LABEL ----->  
      *COLHDG  
      *DESC  
      *NOID  
  
>-- IDENT_ATR ----- *DEFAULT -----  
->  
      *NONE  
      *HI *RI *UL (3 maximum)  
  
>-- DOWN_SEP ----- *DESIGN -----  
->  
      decimal value  
  
>-- ACROSS_SEP --- *DESIGN -----
```

->

decimal value

>-- AT_LOC ----- *CENTRE ----->
or *QUAD1 *QUAD2 *QUAD3 *QUAD4
or *UPPER *LOWER *LEFT *RIGHT
or (row number column number)

>-- WITH_SIZE ---- *AUTO ----->
or (width length)

>-- PANEL_ID ----- *AUTO ----->
or *NONE
or panel identifier

>-- PANEL_TITL --- *FUNCTION -----

-->

or 'Panel title'

>-- BROWSELIST --- *NONE -----

>

name of list

>-- EXIT_KEY ----- *YES -- *EXIT -
- *HIGH - *NONE ->
*NO *MENU *LOW condition
*NEXT
*RETURN
label

>-- MENU_KEY ----- *YES -- *MENU ---
- *NONE ----->
*NO *EXIT condition
*NEXT
*RETURN
label

>-- ADD_KEY ----- *NO ---- *NEXT --- *NONE ---
----->

```

                *YES *RETURN condition
                label

>-- CHANGE_KEY --- *NO ---- *NEXT --
- *NONE ----->
                *YES *RETURN condition
                label

>-- DELETE_KEY --- *NO ---- *NEXT --- *NONE -
----->
                *YES *RETURN condition
                label

>-- PROMPT_KEY --- *DFT --- *AUTO --
- *NONE ----->
                *YES *NEXT condition
                *NO *RETURN
                label

>-- USER_KEYS --- fnc key--'desc'--*NEXT -
- *NONE ->
                |           *RETURN cond |
                |           label      |
                |                       |
                ----- 5 maximum -----

>-- SHOW_NEXT ---- *PRO -----
>
                *YES
                *NO

>-- TEXT ----- 'text' --- line/ --- position -->
                |           row      column |
                ----- 50 max -----
                *TMAPnnn 1 1 (special value)

>-- CURSOR_LOC --- *NONE ----- *NONE -----
----->
                *ATFIELD field name

```

row value column value

>-- *STD_HEAD* ----- **DFT* ----->

**YES*

**NO*

>-- *IGCCNV_KEY* -- **AUTO* -----

**YES*

**NO*

condition name

7.76.1 POP_UP Parameters

ACROSS_SEP
ADD_KEY
AT_LOC
BROWSELIST
CHANGE_KEY
CURSOR_LOC
DELETE_KEY
DESIGN
DOWN_SEP
EXIT_KEY
FIELDS
IDENT_ATR
IDENTIFY
IGCCNV_KEY
MENU_KEY
PANEL_ID
PANEL_TITL
PROMPT_KEY
STD_HEAD
TEXT
USER_KEYS
WITH_SIZE

FIELDS

Specifies either the field(s) that are to be displayed in the pop up window or the name of a group that specifies the field(s). Alternatively, an expandable group expression can be entered in this parameter.

Portability Visual LANSA has multi-page and field spanning line
Considerations restrictions:

Multi-page data (i.e. if the screen format is larger than one page) can be displayed in a Web browser window but NOT in

a LANSAs function.

If a process containing multi-page data is compiled, a warning will be issued if the process is WEB/XML enabled. If the process is NOT WEB/XML enabled, a full function check error will be issued.

Field spanning (i.e. when the field is larger than one line on the screen) is not supported - only a single line will be displayed. No error or warning is issued.

DESIGN

Specifies the design/positioning method which should be used for fields that do not have specific positioning attributes associated with them.

*ACROSS, which is the default value, indicates that fields should be designed "across" the window.

*DOWN indicates the fields should be designed "down" the window in a column.

Note: The default value of this parameter is different to the equivalent default used on DISPLAY or REQUEST commands.

IDENTIFY

Specifies the default identification method to be used for fields that do not have specific identification attributes associated with them.

*LABEL, which is the default value, indicates that fields should be identified by their associated labels in the window.

*DESC indicates that fields should be identified by their associated descriptions in the window.

*COLHDG indicates that fields should be identified by their associated column headings in the window.

*NOID indicates that no identification of the field is required. Only the field itself should be included into the window.

Note: The default value of this parameter is different to the equivalent default used on DISPLAY or REQUEST commands.

IDENT_ATR

Specifies display attributes that are to be associated with identification text (labels, column headings, descriptions, etc) that are displayed in the window.

*DEFAULT, which is the default value, indicates that the system defaults for

identification display attributes should be adopted. They are set up in the system definition block as overall system default values. Refer to [The System Definition Data Areas](#) in the *LANSA for i User Guide* for more details of the system definition block and how to change it.

*NONE indicates that identification text should have no special display attributes associated with it.

Otherwise, specify one or more of the values: *HI (high intensity), *RI (reverse image) and *UL (underline).

This parameter is ignored in **SAA/CUA processes in SAA/CUA** compliant partitions. In such partitions the attributes are determined from the partition wide standards for labels and column headings.

DOWN_SEP

Specifies the spacing between rows on the display that should be used when automatically designing a screen. The value specified must be *DESIGN or a number in the range 1 to 10. Refer to the table in the Comments/Warnings section, for details of what value *DESIGN is actually specifying.

ACROSS_SEP

Specifies the spacing between columns on the display that should be used when automatically designing a screen. The value specified must be *DESIGN or a number in the range 1 to 10. Refer to the table in the Comments/Warnings section for details of what value *DESIGN is actually specifying.

AT_LOC

Specifies the position of the window on the screen panel. It may be entered as a special value that automatically specifies the row and column number according to the table below, or as actual row and column numbers. Values specified are fixed and cannot be varied at execution time. The special values and the actual row and column numbers used are as follows:

Special Value	Row Number	Column Number
*CENTRE	8	22
*QUAD1	2	3
*QUAD2	2	43
*QUAD3	14	3

*QUAD4	14	43
*UPPER	2	3
*LOWER	14	3
*LEFT	2	3
*RIGHT	2	43

WITH_SIZE

Specifies the size (ie: width and length) of the window. It may be entered as a special value *AUTO that automatically computes a width and length, or as actual width and length values.

Values specified are fixed and cannot be varied at execution time.

To use the WITH_SIZE parameter, the AT_LOC parameter must be set to numeric values. If the AT_LOC parameter is unspecified it will default to a special value, and the WITH_SIZE parameter values will be ignored.

When the AT_LOC parameter is one of the following special values, the WITH_SIZE parameter is automatically computed according to the following table, regardless of what value you specify for the parameter.

Special Value	Width	Length
*CENTRE	38	10
*QUAD1	36	10
*QUAD2	35	10
*QUAD3	36	10
*QUAD4	35	10
*UPPER	75	10
*LOWER	75	10
*LEFT	36	22
*RIGHT	35	22

If you specify a specific row and column value for the AT_LOC parameter, and leave the WITH_SIZE parameter as *AUTO, the width and length values will be automatically calculated to fill the entire screen panel to the right of and to the bottom of the nominated locating row and column number.

PANEL_ID

Specifies the identifier that is to be assigned to the panel or pop-up window created by this command.

*AUTO indicates that it should be automatically generated by LANSAs from the function name and the source statement number of the RDML program.

*NONE indicates that no panel identifier is required for this panel or pop-up window.

Otherwise specify a panel identifier from 1 to 10 characters in length. The value specified is fixed and cannot be changed at execution time.

This parameter is valid for SAA/CUA **and** non-SAA/CUA applications.

This parameter is **ignored** if the current partition definition indicates that panel identifiers are never required, no matter what value is specified.

PANEL_TITL

Specifies the title that is to be assigned to the window panel.

*FUNCTION indicates that it should be derived from the RDML function's description.

Otherwise specify a panel title from 1 to 40 characters in length. The value specified is fixed and cannot be changed at execution time.

This parameter is valid for SAA/CUA **and** non-SAA/CUA applications.

BROWSELIST

Specifies the name of a browse list which is also to be included into the window.

*NONE indicates that no browse list is required. The window designed will not have any browse component.

If a browse list is specified it must be defined elsewhere in the RDML program with a DEF_LIST (define list) command.

EXIT_KEY

Specifies the following things about the EXIT function key:

- Whether the EXIT function key is to be enabled.
- What is to happen when the EXIT function key is used.
- In SAA/CUA partitions, which EXIT function key is required.
- A condition to control when the EXIT function key is enabled.

By default the EXIT function key is enabled. To disable the EXIT function key specify *NO as the first value for this parameter.

If the EXIT function key is enabled, you may specify what happens when it is used. The allowable values for this second component of the EXIT_KEY parameter are as follows:

- *EXIT The application should exit completely from LANSAs. (identical to executing an EXIT command).
- *MENU The process's main menu should be re-displayed (identical to executing a MENU command).
- *NEXT Indicates that control should be passed to the next command.
- *RETURN Specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The value *EXIT is the default for this parameter value.

Additionally, in SAA/CUA partitions, you may nominate whether the EXIT function key to be enabled is the "high" exit key or the "low" exit key.

The default value is *HIGH for this parameter value.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

Note: In SAA/CUA applications it is recommended that only the following 2 variations of the EXIT_KEY parameter are used:

EXIT_KEY(*YES *EXIT *HIGH) in a "main program"
EXIT_KEY(*YES *RETURN *LOW) in "subroutines"

MENU_KEY

Specifies whether the MENU function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the MENU key is used.

*YES, which is the default value, indicates that the MENU key should be enabled when the screen is displayed. If *YES is used it is also possible to specify the action to be taken when the menu key is used.

*MENU, the default value, specifies that the process's main menu should be re-displayed.

*EXIT specifies that the application should exit completely from LANSAs.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

*NO indicates that the MENU function key should not be enabled when the screen is displayed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

ADD_KEY

Specifies whether the ADD function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the ADD key is used.

*NO, which is the default value, indicates that the ADD function key should not be enabled when the screen is displayed.

*YES indicates that the ADD key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the ADD key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

CHANGE_KEY

Specifies whether the CHANGE function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the CHANGE key is used.

*NO, which is the default value, indicates that the CHANGE function key should not be enabled when the screen is displayed.

*YES indicates that the CHANGE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the CHANGE key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

DELETE_KEY

Specifies whether the DELETE function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to

happen if the DELETE key is used.

*NO, which is the default value, indicates that the DELETE function key should not be enabled when the screen is displayed.

*YES indicates that the DELETE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the DELETE key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

PROMPT_KEY

Specifies whether the PROMPT function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the PROMPT key is used.

*DFT, which is the default value, indicates that the PROMPT function key should be enabled or disabled automatically according to its default value defined in the system definition data area DC@A01. Refer to [The System Definition Data Area DC@A01](#) in the *LANSA for i User Guide*.

*YES indicates that the PROMPT key should be enabled when the screen is displayed.

*NO indicates that the PROMPT key should NOT be enabled when the screen is displayed.

In any case, when the PROMPT function key is enabled (either by specifying *DFT or *YES for the first part of this parameter), it is possible to also specify what is to happen if the function key is used. Allowable values for this part of the parameter are:

*AUTO indicates that the prompt key processing should be handled

automatically by LANSAs. Refer to [Prompt_Key Processing](#) before attempting to use this option.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

USER_KEYS

Specifies up to 5 additional user function keys that can be enabled when the screen format is displayed at the workstation.

Any user function keys assigned must not conflict with function keys assigned to the standard LANSAs functions of EXIT, MENU, MESSAGES, ADD, CHANGE, DELETE or PROMPT when they are enabled on a command (ie: a function key cannot be assigned to more than one function).

Additional user function keys are specified in the format:

```
(fnc key number) 'description' *NEXT *NONE
*RETURN cond name
label
```

where:

fnc key number Is the function key number in the range 1 to 24 or one of the special values *ROLLUP (roll up key) or *ROLLDOWN (roll down key).

'description' Is a description of the function assigned to the function key. This description will be displayed on line 23 of the screen format. Maximum length is 8 characters.

- *NEXT Is the default and indicates that the next command (after this one) should receive control.
- *RETURN Indicates that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.
- Label Indicates the command label to which control should be passed if the command key is used.
- *NONE Indicates that no condition applies to control when the function key is to be enabled or disabled.
- cond name Indicates that a condition defined by a DEF_COND command should be evaluated to determine whether to enable or disable the function key.

Refer to the IF_KEY command for details of how the function key that was used can be tested in the RDML program.

As an example of usage consider the following:

```
POP_UP FIELDS(#PRODUCT) USER_KEYS((14 'Commit')(15 'Purge'))
  IF_KEY WAS(*USERKEY1)
    << Commit logic >>
  ENDIF
  IF_KEY WAS(*USERKEY2)
    << Purge logic >>
  ENDIF
```

Note that the IF_KEY command refers to the keys by symbolic names that indicate the order they are declared in the USER_KEYS parameter, not the actual function key numbers assigned to them. This makes changing function key assignments easier.

TEXT

Allows the specification of up to 50 "**text strings**" that are to appear on the screen panel or report. Each text string specified is restricted to a maximum length of 20 characters.

When a text string is specified it should be followed by a row/line number and a column/position number that indicates where it should appear on the screen panel or report.

For example:

```
TEXT(('ACME' 6 2)('ENGINEERING' 7 2))
```

specifies 2 text strings to appear at line 6, position 2 and line 7, position 2 respectively.

Portability Considerations In Visual LANSA this parameter should only be edited using the screen or report painter which will replace any text with a text map. DO NOT enter text using the command prompt or free format editor as it will not pass the full function checker if checked in to LANSA for i.

All Platforms

The text map is used by the screen or report design facilities to store the details of all the text strings associated with the screen panel or report lines.

Once a screen or report layout has been "painted" and saved, all text details from the layout are stored in a "text map". The text map is then subsequently changed by using the "painter" again.

The presence of a text map is indicated by a TEXT parameter that looks like this example:

```
TEXT((*TMAPnnn 1 1))
```

where "nnn" is a unique number (within this function) that identifies the stored text map.

Some **very important** things about "text maps" and *TMAPnnn identifiers that you **must** know are:

- Never specify *TMAPnnn identifiers of your own or change *TMAPnnn identifiers to other values. Leave the assignment and management of *TMAPnnn identifiers to the screen and report design facilities.
- When copying a command that has an *TMAPnnn identifier, remove the *TMAPnnn references (ie: the whole TEXT parameter) from the copied command. If you fail to do this, then the full function checker will detect the duplicated use of *TMAPnnn identifiers, and issue a fatal error message before any loss occurs.
- Never remove an *TMAPnnn identifier from a command. If this is done then the associated text map may be deleted, or reused in another command, during a full function check or compilation. Loss of text details is likely to

result.

- Never "comment out" a command that contains a valid *TMAPnnn identifier. This is just another variation of the preceding warning and it runs the same risks of loss or reuse of text.
- Never specify *TMAPnnn values in an Application Template. In the template context *TMAPnnn values have no meaning. Use the "text string" format in commands used in, and initially generated by, Application Templates.

CURSOR_LOC

Specifies any user controlled cursor positioning that is required. The CURSOR_LOC parameter must always contain 2 values, which may take any of the following forms:

*NONE / *NONE: which are the default values indicate that no user controlled cursor positioning is required. Normal LANSAs cursor control is to be used. When a screen is displayed the cursor will be positioned to either the first input capable field or the first field in error.

*ATFIELD / Field name: specifies that the cursor should be positioned to the named field. If the named field is not on the display or a field error exists, normal LANSAs cursor control will be used. Otherwise the cursor will be positioned to the nominated field.

Row value / Column value: specifies that the "values" nominated indicate the row and column number at which the cursor is to be positioned. The "values" nominated may be an alphanumeric literal (e.g.: 15) or the name of a field that contains the value (e.g.: #ROW). In all cases the value must be numeric. If the row or column values are invalid or a field error exists, normal LANSAs cursor control will be used. Otherwise the cursor will be positioned at the row and column specified.

When the row and column option is used **and** the row and column values are specified **as fields** (rather than numeric literals), the row and column number where the cursor was located when the command completed execution will be **returned in them**.

Note: The CURSOR_LOC does not behave in the same way on Windows as on IBM i. On a Windows platform the value retrieved is the first position of the field the cursor is currently in.

This feature is a useful way of retrieving the location of the screen cursor at the time the command completed execution. In cases where you wish to **retrieve** the cursor location, but do not want to **specify it** before output to the screen, use

coding like this:

```
CHANGE FIELD(#ROW #COL) TO(0)  
POP_UP FIELDS(#FIELD1 .. #FIELD10) CURSOR_LOC(#ROW #COL)
```

When the POP_UP command is executed #ROW and #COL are both zero, which is an invalid cursor location. In such cases normal LANSAs cursor control is resumed and the user positioning request is ignored. However, after completion of the command, fields #ROW and #COL will contain the location of the cursor at the time the POP_UP command completed execution.

STD_HEAD

Specifies whether or not the standard LANSAs design for the screen heading lines (lines 1 and 2) should be used.

*DFT, which is the default value, indicates that the system default value for the STD_HEAD parameter should be used. The system default value is stored in the LANSAs system definition block. Refer to [The System Definition Data Areas](#) in the *LANSAs for i User Guide* for details of the system definition block and how to change it.

*YES indicates that the standard LANSAs screen heading lines should be used. When this option is used lines 1 and 2 of the display are not available for the positioning of user fields.

*NO indicates that the standard LANSAs screen heading lines should not be used. In this case lines 1 and 2 of the display can be used to position user fields.

IGCCNV_KEY

Controls the appearance of the text "Fnn=XXXXXX" in the function key area, of the function key assigned to support IGC conversion.

This parameter is **ignored** if the language under which this function is being compiled does not have the "IGCCNV required" flag enabled, or if this function uses the *NOIGCCNV options keyword (refer to the FUNCTION command).

Also note that this parameter only controls the appearance of the text "Fnn=XXXXXX" in the function key area. It does **not** control the enablement of the IGCCNV DDS keyword in the display file associated with this function. This is controlled by the setting of the "IGCCNV required" flag and the use of the *NOIGCCNV option.

*AUTO, which is the default value, indicates that appearance of the function key text should be determined automatically. The automatic rules used to

determine whether or not to show the function key text are:

- If there are no fields with keyboard shift J, E or O involved, the text will not appear (ignore all following rules).
- For a REQUEST command the text will always appear.
- For DISPLAY or POP_UP commands, the current "mode" is tested. If the mode is "change" (ie: fields on the screen are input capable), the text will appear. For all other modes the text will not appear.

Other allowable values for this parameter are *YES, indicating that the text should always appear, or, *NO indicating that the text should never appear.

The final option allows the nomination of a condition previously defined by a DEF_COND command. If the condition is true the text should appear. If the condition is false, the text should not appear.

Portability When used with Visual LANSA, this parameter is ignored
Considerations with no known effect to the application.

7.76.2 POP_UP Comments / Warnings

POP_UP windows are defined and presented via conventional DDS (Data Description Specifications). They **do not use** "user defined data streams" and thus do not, and will not ever have, upward compatibility problems.

The POP_UP command is a "mode sensitive" command. For details of "mode sensitive" command processing, refer to [Screen Modes and Mode Sensitive Commands](#).

When a pop up window appears, all fields already on the screen are protected and cannot be changed by the user.

Only the function keys enabled by the POP_UP window command are enabled. Function keys enabled on the panel that it overlaid are disabled and cannot be used.

A pop-up window is a "miniature" screen panel, and is laid out exactly like a full screen panel. Consider the following:

```

----- l : window location (AT_LOC) specified
|
|           as a row and column number.
|
| BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|
--->| iiiiii   ttttttttttttttttttttt |      B
    B                |      B
    B                |      B
    B                length  B
    B                |      B
B<----- width -----|----->B
    B                |      B
    B                |      B
    B                |      B
B MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
B FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF | FFFFFFFF B
B FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF | FFFFFFFF B
    B                |      B
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|

```

* where: "iiiiiii" is the panel identifier.
 "ttt...tt" is the panel title.

"MMM...MM" is the message line.
"FFFFFFF" are the 2 function key lines.
"BB...BB" are the window border fill characters.

Note that the panel identifier and title line may be omitted. The message line and function key lines may appear in a different order if specified for the partition. In non-SAA/CUA functions only one line of function keys is presented.

The window location is specified as a row and column number and is used to position the upper left hand corner (within the border) of the window. Since the window border must also appear, the minimum row value is 2 and the maximum is 23. Likewise the minimum column number is 3 and the maximum 77.

The pop up window border is presented in SAA/CUA applications according to the partition level definition of panel element category PBWB (normally reverse video blue). In non-SAA/CUA applications the value used is reverse video green.

The pop up window width and length are specified in character positions and nominate the dimensions within (but not including) the border.

Rows within the window used for panel identification, messages and function keys are reserved and you cannot position fields anywhere on these rows.

The first and last 2 columns within a pop up window are reserved and you cannot position a field to start, end or span one of these columns.

When a field is positioned into a pop up window, the field (including its label or description) and all leading and trailing attribute bytes must entirely fit onto one line of the window. A field cannot span lines of the display in a pop up window.

Pop up windows can be used for input or output operations just like the DISPLAY command. Note also that the POP_UP command is "mode sensitive" just like the DISPLAY command.

Pop up windows can contain a browse list. The browse list may be used for input or output operations.

When specifying field locations within a window, note that the positions are relative to the window location. Thus:

```
POP_UP FIELDS((#DATE *R2 *C3))
```

specifies that DATE is to be positioned at row 2, column 3 **within** the pop up window.

When using the CURSOR_LOC parameter to nominate a specific row and

column for cursor positioning, note that the positions are absolute. The values specified relate to the entire screen panel, not the window. Thus values outside the pop up window borders may be specified.

If you want the name of the field in which the CURSOR was located when Enter or any other AID was pressed, to be returned to your function, then refer to a field named #CURLOC\$FN within your function.

#CURLOC\$FN (alpha, 10) will contain the name of the field.

The following table indicates all combinations of the DESIGN and IDENTIFY parameters and what values result when the *DESIGN default is used in the associated DOWN_SEP or ACROSS_SEP parameters:

Specified: DESIGN	Specified: IDENTIFY	*DESIGN Specified: DOWN_SEP	*DESIGN Specified: ACROSS_SEP
*DOWN	*COLHDG	5	1
*DOWN	*LABEL	1	1
*DOWN	*DESC	1	1
*DOWN	*NOID	1	1
*ACROSS	*COLHDG	5	1
*ACROSS	*LABEL	1	1
*ACROSS	*DESC	1	1
*ACROSS	*NOID	1	1

In some cases all the fields specified in the FIELDS parameter will not fit on one screen. In this case a second, third, fourth, etc. window is automatically designed as required.

In terms of the RDML program they can be treated like one "long" window. LANSAs will automatically process the windows one after another until they have all been processed. When all windows have been processed the next RDML command is executed.

So when you use the POP_UP command you may in fact be requesting that 2 or 3 or more windows be displayed one after another.

This facility is a feature of the automatic design procedures. If you are coding the RDML program yourself it is **advisable** to "split up" the POP_UP command into multiple POP_UP commands that have only one window format each or increase the size of the window.

Note that format control characters in 2nd level message text have no effect when the message is displayed from a POP_UP.

7.76.3 POP_UP Examples

The following examples apply to the POP_UP command. For further examples refer to the DISPLAY command which is functionally identical.

Example 1: Present fields #ORDNUM, #CUSTNUM and #DATEDUE to the user in a pop up window located in the center of the screen:

```
POP_UP  FIELDS(#ORDNUM #CUSTNUM #DATEDUE)
```

or, identically:

```
GROUP_BY  NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #I  
POP_UP  FIELDS(#ORDERHEAD)
```

Example 2: Display an order, including all line details in a scrollable area, in a pop up window:

```
GROUP_BY  NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #I  
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA  
  
POP_UP  FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE) AT_LC
```

Example 3: Request #ORDNUM #CUSTNUM and #DATEDUE and also specify specific positions and identification methods as field attributes.

For details of field attributes, refer to [Field Attributes and their Use](#).

When specific positions for a field are nominated, the automatic design facility is effectively "disabled". For example,

```
GROUP_BY  NAME(#ORDERHEAD) FIELDS((#ORDNUM *COLHDG *I  
  
POP_UP  FIELDS(#ORDERHEAD) AT_LOC(*UPPER) TEXT(('--DATE--  
' 6 37))
```

would cause a popup something like this to be designed:

```
-----  
:                               :  
: Company   Customer no:      :  
: Order                               :  
: Number                               :  
:                               --DATE-- :  
:
```

Note that the manual specification of row and column numbers and "text" is not required. The screen design facility can be used to modify an "automatic" design much more quickly and easily. For details, refer [The Screen Design Facility](#) in the *LANSA for i User Guide*.

After the screen design facility has been used on a POP_UP command, the associated FIELDS parameter (in the POP_UP or GROUP_BY command) will be automatically re-written with the required row, column and method of identification attributes.

Remember that if an expandable group expression was used, LANSA will substitute the expression with the fields that constitute it.

In addition, the TEXT parameter of the POP_UP command will also be automatically re-written.

Example 4: Use an Expandable Group expression and redesign the layout using the screen design facility:

```
GROUP_BY  NAME(#XG_ORDHDG) FIELDS(#ORDNUM #CUSTNUM #
POP_UP    FIELDS(#XG_ORDHDG) DESIGN(*ACROSS) IDENTIFY(*CC
```

The popup designed automatically would look like this:

```
-----
:                               :
: Company Order  Date          :
: Order  Customer Order        :
: Number  Number  Due           :
: _____  _____  _____ :
:                               :
:                               :
-----
```

If the layout is changed using the screen design facility to look like this:

```
Company Order
Order  Customer
Number  Number
_____  _____  Date Order Due _____
```

then the POP_UP command FIELDS parameter will be expanded as follows:

POP_UP FIELDS((#ORDNUM *L2 *P3) (#CUSTNUM *L2 *P13) (#DATI

7.77 PRINT

The PRINT command is used to either test the "trigger" of all break lines and/or print one or more detail lines.

Break lines (see DEF_BREAK command) are printed only when their associated "trigger" has been satisfied and thus it is necessary to cause LANSAs to check the "trigger" values at the appropriate time. If their "trigger" has been satisfied the break line(s) will be printed.

Detail lines (see DEF_LINE command) are not printed automatically or "triggered" like break lines (see DEF_BREAK command), heading lines (see DEF_HEAD command) or foot lines (see DEF_FOOT command). To print a detail line you must request that it be printed by executing a PRINT command.

Review [Producing Reports Using LANSAs](#) before attempting to use a PRINT command.

Also See

[7.77.1 PRINT Parameters](#)

[7.77.2 PRINT Examples](#)

Optional

```
PRINT ----- LINE ----- *BREAKS -----
```

```
->
```

```
list of DEF_LINE group names
```

```
|           |  
----- 20 max -----
```

```
>-- ON_REPORT ---- 1 -----|
```

```
report number 1 -> 8
```

7.77.1 PRINT Parameters

LINE

ON_REPORT

LINE

Specifies what is to be printed by this command.

*BREAKS, which is the default value, indicates that there are no detail lines (DEF_LINE commands) to be printed, but all break lines (DEF_BREAK commands) should be checked and printed if their "trigger" has been satisfied.

Otherwise specify from 1 to 20 detail line group names previously defined with DEF_LINE commands. When this option is used all break line "triggers" will also be tested implicitly.

ON_REPORT

Specifies the report on which the line groups nominated are to be printed. Up to 8 reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this report is report number 1.

7.77.2 PRINT Examples

Example 1: Write an RDML program to read a regional sales file and print details of each record read.

```
DEF_LINE  NAME(#DETAIL) FIELDS(#REGION #PRODES #VALUE)

SELECT   FIELDS(#DETAIL) FROM_FILE(SALEHIST)
PRINT   LINE(#DETAIL)
ENDSELECT

ENDPRINT
```

Example 2: Write an RDML program to read a regional sales file, print details of each record read and produce regional subtotals.

```
DEF_LINE  NAME(#DETAIL) FIELDS(#REGION #PRODES #VALUE)
DEF_BREAK NAME(#REGTOT) FIELDS(#REGVAL) TRIGGER_BY(#RE

SELECT   FIELDS(#DETAIL) FROM_FILE(SALEHIST)
KEEP_TOTAL OF_FIELD(#VALUE) IN_FIELD(#REGVAL) BY_FIELD(#R
PRINT   LINE(#DETAIL)
ENDSELECT

ENDPRINT
```

Example 3: Write an RDML program to read a regional sales file and print the regional subtotals only (ie: a summary report).

```
DEF_BREAK NAME(#REGTOT) FIELDS(#REGION #REGVAL) TRIGGE
SELECT   FIELDS(#REGION #VALUE) FROM_FILE(SALEHIST)
KEEP_TOTAL OF_FIELD(#VALUE) IN_FIELD(#REGVAL) BY_FIELD(#R
PRINT   LINE(*BREAKS)
ENDSELECT

ENDPRINT
```

7.78 RANGECHECK

The RANGECHECK command is used to check a field against one or more ranges of values.

Also See

[7.78.1 RANGECHECK Parameters](#)

[7.78.2 RANGECHECK Examples](#)

[7.4 BEGINCHECK](#)

[7.7 CALLCHECK](#)

[7.14 CONDCHECK](#)

[7.16 DATECHECK](#)

[7.35 ENDCHECK](#)

[7.48 FILECHECK](#)

[7.99 VALUECHECK](#)

Required

```
RANGECHECK --- FIELD ----- field name -----  
----->
```

```
>-- RANGE ----- low value -- high value ----->  
      |           |  
      ----- 20 max -----
```

Optional

```
>-- IN_RANGE ----- *NEXT ----->  
      *ERROR  
      *ACCEPT
```

```
>-- OUT_RANGE ---- *ERROR -----  
->  
      *NEXT  
      *ACCEPT
```

```
>-- MSGTXT ----- *NONE ----->
```

message text

>-- *MSGID* ----- *DCU0001* ----->
message identifier

>-- *MSGF* ----- *DC@M01 . *LIBL* -----
>
message file . library name

>-- *MSGDTA* ----- *substitution variables* -----|
|*expandable group expression* |
----- *20 max* -----

7.78.1 RANGECHECK Parameters

FIELD

IN_RANGE

MSGDTA

MSGF

MSGID

MSGTXT

OUT_RANGE

RANGE

FIELD

Specifies the name of the field which is to be checked.

RANGE

Specifies from 1 to 20 ranges of values that are to be checked against the field. Each individual range must consist of a "low" value and a "high" value. See following examples for more details.

IN_RANGE

Specifies the action to be taken if the field is found to be in one (or more) of the ranges specified in the RANGE parameter.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

OUT_RANGE

Specifies the action to be taken if the field is not found to be in any of the range(s) specified in the RANGE parameter.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this:

```
MSGDTA('BOLTS   ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA('"BOLTS   "' #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.78.2 RANGECHECK Examples

[Structuring Functions for Inline Validation](#)

[Structuring Functions to Use a Validation Subroutine](#)

[Using the RANGECKECK Command for Inline Validation](#)

[Using the RANGECHECK Command for Validation with a Subroutine](#)

Structuring Functions for Inline Validation

Typically functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for inline validation like this:

```
BEGIN_LOOP
REQUEST  << INPUT >>
BEGINCHECK
    << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK
    << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is passed back to the REQUEST command. This happens because of the default IF_ERROR(*LASTDIS) parameter on the ENDCHECK command.

Structuring Functions to Use a Validation Subroutine

Typically functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for subroutine validation like this:

```
DEFINE  FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND  NAME(*NOERRORS) COND('#ERRORCNT = 0')

BEGIN_LOOP
DUNTIL  COND(*NOERRORS)
REQUEST  << INPUT >>
EXECUTE  SUBROUTINE(VALIDATE)
ENDUNTIL
```

```
        << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE   FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
        << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is returned to the main function loop with #ERRORCNT > 0.

Using the RANGECHECK Command for Inline Validation

This example demonstrates how to use the RANGECHECK command within the main program block to check that an employee number is within a range of values.

```
DEF_LIST  NAME(#EMPBROWSE) FIELDS(#EMPNO #GIVENAME #SU
*
BEGIN_LOOP
REQUEST  FIELDS(#EMPNO #GIVENAME #SURNAME) BROWSELIST
*
BEGINCHECK
RANGECHECK FIELD(#EMPNO) RANGE((A0000 A9999)) MSGTXT('Em
ENDCHECK
*
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

If the value of #EMPNO is outside the range of A0000 to A9999 the message defined with the RANGECHECK command is issued and program control returns to the last screen displayed. In this case the last screen displayed is the REQUEST screen.

Using the RANGECHECK Command for Validation with a Subroutine

This example demonstrates how to use the RANGECHECK command inside a subroutine to check an employee number is within a range of values.

After the user enters the requested details the VALIDATE subroutine is called. It checks that the value of #EMPNO is within the range of A0000 to A9999. If this is not true the message defined in the RANGECHECK command is given and the DOUNTIL loop executes again. When a value for #EMPNO is entered that is within the specified range the DOUNTIL loop ends and processing of the verified input is done.

```
DEFINE FIELD(#ERRORCNT) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #GIVENAME #SURNAME)
*
BEGIN_LOOP
DOUNTIL COND(*NOERRORS)
REQUEST FIELDS(#EMPNO #GIVENAME #SURNAME) BROWSELIST
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
*
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
*
SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
*
BEGINCHECK KEEP_COUNT(#ERRORCNT)
RANGECHECK FIELD(#EMPNO) RANGE((A0000 A9999)) MSGTXT('Empno out of range')
ENDCHECK IF_ERROR(*NEXT)
*
ENDROUTINE
```

7.79 RENAME

The RENAME command is used to rename a field in a file that is referenced by the function.

The RENAME command is most commonly used when referencing 2 or more files that contain a field with the same name but a different "meaning" or "value". By renaming the field in one or more of the files it is easier to reference the individual fields in all of the files.

Note that the rename only occurs within the function involved and has no effect upon the file definition or data dictionary.

The current release of LANSAs does not allow the RENAME command to be used in functions that use the *DBOPTIMISE features. Refer to the FUNCTION command for more information about this feature.

Portability Considerations Refer to parameter [FROM_FILE](#).

Also See

[7.79.1 RENAME Parameters](#)

[7.79.2 RENAME Comments / Warnings](#)

[7.79.3 RENAME Examples](#)

Required

```
RENAME ----- FIELD ----- field name -----  
->  
  
    >-- FROM_FILE ---- file name . *FIRST -----  
>  
        library name  
  
    >-- WITH_NAME ---- new field name -----  
|
```

7.79.1 RENAME Parameters

FIELD

FROM_FILE

WITH_NAME

FIELD

Specifies the field (in the file nominated in the FROM_FILE parameter) that is to be renamed. The field specified must start with a # and must be defined in the file nominated.

FROM_FILE

Refer to [Specifying File Names in I/O commands](#).

WITH_NAME

Specifies the new name that is to be assigned to the field specified in the field parameter. The new name must start with a # and must not be the name of another field in the LANSAs data dictionary, or the name of a group or a list.

7.79.2 RENAME Comments / Warnings

In terms of processing, when a field is renamed it appears exactly as if the new name is the name of the field in the file. Attempting to use the "old" name in any context will probably fail to produce the desired results.

7.79.3 RENAME Examples

The following example applies to the RENAME command.

A function uses 2 customer master files called CUSMST1 and CUSTMST2. Both contain field CUSTNO. Rename CUSTNO in CUSMST2 so that it is called CUSTNO2:

```
RENAME FIELD(#CUSTNO) FROM_FILE(CUSTMST2) WITH_NAME(#C
```

7.80 REQUEST

The REQUEST command allows the user to input information at a workstation. The REQUEST command is only valid in RDMLX functions when being used on the Web. If it is used elsewhere a fatal error occurs at runtime. If this occurs, either put your REQUEST command in an RDML function or use a Form to show user information.

Portability Considerations

Refer to parameters: [FIELDS](#),
[IGCCNV_KEY](#) , [OPTIONS](#) , [SHOW_NEXT](#) and
[TEXT](#).

Also See

[7.80.1 REQUEST Parameters](#)

[7.80.2 REQUEST Comments / Warnings](#)

[7.80.3 REQUEST Examples](#)

Optional

```
REQUEST ----- FIELDS -----  
- field name field attributes --->  
      |           |           ||  
      |           --- 7 max ----- |  
      | expandable group expression |  
      |----- 1000 max for RDMLX-----|  
      ----- 100 max for RDML -----  
  
>-- DESIGN ----- *IDENTIFY ----->  
      *DOWN  
      *ACROSS  
  
>-- IDENTIFY ----- *DESIGN ----->  
      *COLHDG  
      *LABEL  
      *DESC  
      *NOID  
  
>-- IDENT_ATR ----- *DEFAULT -----  
->
```

**NONE*
**HI *RI *UL (3 maximum)*

*>-- DOWN_SEP ----- *DESIGN -----*
->
decimal value

*>-- ACROSS_SEP --- *DESIGN -----*
->
decimal value

*>-- BROWSELIST --- *NONE ----- 999 -----*
->
name of list no.entries/page

*>-- EXIT_KEY ----- *YES -- *EXIT -*
*- *HIGH - *NONE ->*
**NO *MENU *LOW condition*
**NEXT*
**RETURN*
label

*>-- MENU_KEY ----- *YES -- *MENU ---*
*- *NONE ----->*
**NO *EXIT condition*
**NEXT*
**RETURN*
label

*>-- ADD_KEY ----- *NO ---- *NEXT --- *NONE ---*
----->
**YES *RETURN condition*
label

*>-- CHANGE_KEY --- *NO ---- *NEXT --*
*- *NONE ----->*
**YES *RETURN condition*
label

>-- DELETE_KEY --- *NO ---- *NEXT --- *NONE -
----->

*YES *RETURN condition
label

>-- PROMPT_KEY --- *DFT --- *AUTO --
- *NONE ----->

*YES *NEXT condition
*NO label

>-- USER_KEYS --- fnc key--'desc'--*NEXT -
- *NONE ->

	*RETURN cond	
	label	

----- 5 maximum -----

>-- PANEL_ID ----- *AUTO ----->
or *NONE
or panel identifier

>-- PANEL_TITL --- *FUNCTION -----
-->
or 'Panel title'

>-- SHOW_NEXT ---- *PRO -----
>
*YES
*NO

>-- TEXT ----- 'text' --- line/ --- position -->
| row column |
----- 50 max -----
*TMAPnnn 1 1 (special value)

>-- CURSOR_LOC --- *NONE ----- *NONE -----
----->
*ATFIELD field name
row value column value

>-- STD_HEAD ----- *DFT ----->
 *YES
 *NO

>-- OPTIONS ----- *NONE ----->
 *NOREAD *OVERLAY (2 maximum)

>-- IGCCNV_KEY -- *AUTO -----
 *YES
 *NO
 condition name

7.80.1 REQUEST Parameters

ACROSS_SEP
ADD_KEY
BROWSELIST
CHANGE_KEY
CURSOR_LOC
DELETE_KEY
DESIGN
DOWN_SEP
EXIT_KEY
FIELDS
IDENT_ATR
IDENTIFY
IGCCNV_KEY
MENU_KEY
OPTIONS
PANEL_ID
PANEL_TITL
PROMPT_KEY
SHOW_NEXT
STD_HEAD
TEXT
USER_KEYS

FIELDS

Specifies either the field(s) that are to be input at the workstation or the name of a group that specifies the field(s) to be input. An expandable group expression is allowed in this parameter.

Portability Visual LANSA has multi-page and field spanning line
Considerations restrictions:

Multi-page data (i.e. if the screen format is larger than one page) can be displayed in a Web browser window but NOT in

a LANSAs function.

If a process containing multi-page data is compiled, a warning will be issued if the process is WEB/XML enabled. If the process is NOT WEB/XML enabled, a full function check error will be issued.

Field spanning (i.e. when the field is larger than one line on the screen) is not supported - only a single line will be displayed. No error or warning is issued.

DESIGN

Specifies the design/positioning method which should be used for fields that **do not have specific positioning attributes** associated with them.

*IDENTIFY, which is the default value, indicates that the design method should be the default method associated with the IDENTIFY parameter. Refer to the table in the comments section for more details.

*DOWN indicates that the fields should be designed "down" the screen in a column.

*ACROSS indicates that fields should be designed "across" the screen in a row.

IDENTIFY

Specifies the default identification method to be used for fields that **do not have a specific identification attribute** associated with them.

*DESIGN, which is the default value, indicates that the fields should be identified with by the default method associated with the DESIGN parameter. See the table in the comments section for more details.

*LABEL indicates that fields should be identified by their associated labels on the screen.

*DESC indicates that fields should be identified by their associated descriptions on the screen.

*COLHDG indicates that fields should be identified by their associated column headings on the screen.

*NOID indicates that no identification of the field is required. Only the field itself should be included into the screen design.

IDENT_ATR

Specifies display attributes that are to be associated with identification text (labels, description, column headings, etc) that are displayed on the screen.

*DEFAULT, which is the default value, indicates that the system defaults for identification display attributes should be adopted. These are set up in the system definition block as overall system default values. Refer to [The System Definition Data Areas](#) in the *LANSAs for i User Guide* for more details of the system definition block and how to change it.

*NONE indicates that identification text should have no special display attributes associated with it.

Otherwise, specify one or more of the values: *HI (high intensity), *RI (reverse image) and *UL (underline).

This parameter is **ignored in SAA/CUA processes in SAA/CUA compliant partitions**. In such partitions the attributes are determined from the partition wide standards for labels and column headings.

DOWN_SEP

Specifies the spacing between rows on the display that should be used when automatically designing a screen. The value specified must be *DESIGN or a number in the range 1 to 10. Refer to the table in the comments section for details of what value *DESIGN is actually specifying.

ACROSS_SEP

Specifies the spacing between columns on the display that should be used when automatically designing a screen. The value specified must be *DESIGN or a number in the range 1 to 10. Refer to the table in the comments section for details of what value *DESIGN is actually specifying.

BROWSELIST

Specifies the name of a browse list which is also to be included into the screen format, and optionally, the number of entries of the browse list that should appear in the screen panel.

*NONE indicates that no browse list is required. The screen designed will not have any browse component.

If a browse list is specified, then you may also specify the number of entries from the browse list that are to appear on the screen panel. This may leave space below the browse list for other details (which can be overlaid by a subsequent screen). The default of 999 entries indicates that the browse list should extend to the bottom of the screen panel.

If a browse list is specified it must be defined elsewhere in the RDML program with a DEF_LIST (define list) command.

EXIT_KEY

Specifies the following things about the EXIT function key:

- Whether the EXIT function key is to be enabled.
- What is to happen when the EXIT function key is used.
- In SAA/CUA partitions, which EXIT function key is required.
- A condition to control when the EXIT function key is enabled.

By default the EXIT function key is enabled. To disable the EXIT function key specify *NO as the first value for this parameter.

If the EXIT function key is enabled, you may specify what happens when it is used. The allowable values for this second component of the EXIT_KEY parameter are as follows:

- *EXIT The application should exit completely from LANSAP. (identical to executing an EXIT command).
- *MENU The process's main menu should be re-displayed. (identical to executing a MENU command).
- *NEXT Indicates that control should be passed to the next command.
- *RETURN Specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The value *EXIT is the default for this parameter value.

Additionally, in SAA/CUA partitions, you may nominate whether the EXIT function key to be enabled is the "high" exit key or the "low" exit key.

The default value is *HIGH for this parameter value.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

Note: In SAA/CUA applications it is recommended that only the following 2 variations of the EXIT_KEY parameter are used:

EXIT_KEY(*YES *EXIT *HIGH) in a "main program"
EXIT_KEY(*YES *RETURN *LOW) in "subroutines"

MENU_KEY

Specifies whether the MENU function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the MENU key is used.

*YES, which is the default value, indicates that the MENU key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the MENU key is used.

*MENU, the default value, specifies that the main menu of the process should be redisplayed.

*EXIT specifies that the application should exit completely from LANSAs.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

*NO indicates that the MENU function key should not be enabled when the screen is displayed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

ADD_KEY

Specifies whether the ADD function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the ADD key is used.

*NO, which is the default value, indicates that the ADD function key should not be enabled when the screen is displayed.

*YES indicates that the ADD key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the ADD key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

CHANGE_KEY

Specifies whether the CHANGE function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the CHANGE key is used.

*NO, which is the default value, indicates that the CHANGE function key should not be enabled when the screen is displayed.

*YES indicates that the CHANGE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the CHANGE key is used.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program

by a DEF_COND (define condition) command.

DELETE_KEY

Specifies whether the DELETE function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the DELETE key is used.

*NO, which is the default value, indicates that the DELETE function key should not be enabled when the screen is displayed.

*YES indicates that the DELETE key should be enabled when the screen is displayed. If *YES is used it is also possible to nominate a command label to which control should be passed when the DELETE key is used

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

PROMPT_KEY

Specifies whether the PROMPT function key is to be enabled when this screen format is displayed at the workstation. In addition it also specifies what is to happen if the PROMPT key is used.

*DFT, which is the default value, indicates that the PROMPT function key should be enabled or disabled automatically according to its default value defined in the system definition data area DC@A01. Refer to [The System Definition Data Area DC@A01](#) in the *LANSA for i User Guide* for details of the DC@A01 system definition data area.

*YES indicates that the PROMPT key should be enabled when the screen is displayed.

*NO indicates that the PROMPT key should NOT be enabled when the screen is displayed.

In any case, when the PROMPT function key is enabled (either by specifying *DFT or *YES for the first part of this parameter), it is possible to also specify what is to happen if the function key is used. Allowable values for this part of the parameter are:

*AUTO indicates that the prompt key processing should be handled automatically by LANSAs. Refer to [Prompt_Key Processing](#) before attempting to use this option.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The final value that can be specified for this parameter allows a condition to be named to control when the function key should be enabled. The default value is *NONE that indicates no condition should apply. The function key will be enabled according to the normal rules.

If a condition name is specified it must be defined within the RDML program by a DEF_COND (define condition) command.

USER_KEYS

Specifies up to 5 additional user function keys that can be enabled when the screen format is displayed at the workstation.

Any user function keys assigned must not conflict with function keys assigned to the standard LANSAs functions of EXIT, MENU, MESSAGES, ADD, CHANGE, DELETE or PROMPT when they are enabled on a command (ie: a function key cannot be assigned to more than one function).

Additional user function keys are specified in the format:

```
(fnc key number) 'description' *NEXT *NONE
                    *RETURN Cond name
                    label
```

where

fnc key Is the function key number in the range 1 to 24 or one of the special

number values *ROLLUP (roll up key) or *ROLLDOWN (roll down key).

'description' Is a description of the function assigned to the function key. This description will be displayed on line 23 of the screen format. Maximum length is 8 characters.

*NEXT Is the default and indicates that the next command (after this one) should receive control.

*RETURN Indicates that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

Label Indicates the command label to which control should be passed if the command key is used.

*NONE Indicates that no condition applies to control when the function key is to be enabled or disabled.

Cond name Indicates that a condition defined by a DEF_COND command should be evaluated to determine whether to enable or disable the function key.

Refer to the IF_KEY command for details of how the function key that was used can be tested in the RDML program.

As an example of usage consider the following:

```
DISPLAY FIELDS(#PRODUCT) USER_KEYS((14 'Commit')(15 'Purge'))
  IF_KEY WAS(*USERKEY1)
    << Commit logic >>
  ENDIF
  IF_KEY WAS(*USERKEY2)
    << Purge logic >>
  ENDIF
```

Note that the IF_KEY command refers to the keys by symbolic names that indicate the order they are declared in the USER_KEYS parameter, not the actual function key numbers assigned to them. This makes changing function key assignments easier.

PANEL_ID

Specifies the identifier that is to be assigned to the panel or pop-up window

created by this command.

*AUTO indicates that it should be automatically generated by LANSAs from the function name and the source statement number of the RDML program.

*NONE indicates that no panel identifier is required for this panel or pop-up window.

Otherwise specify a panel identifier from 1 to 10 characters in length. The value specified is fixed and cannot be changed at execution time.

This parameter is valid for **SAA/CUA** applications only.

This parameter is **ignored** if the current partition definition indicates that panel identifiers are never required, no matter what value is specified.

PANEL_TITL

Specifies the title that is to be assigned to the window panel.

*FUNCTION indicates that it should be derived from the RDML function's description.

Otherwise specify a panel title from 1 to 40 characters in length. The value specified is fixed and cannot be changed at execution time.

This parameter is valid for **SAA/CUA applications only**.

SHOW_NEXT

Specifies whether the "next function" field should be shown on line 22 of the screen. The next function field is facility that allows transfer between the functions in a process without the need to return to the process menu each time. Refer to [The Process Control Table](#) in the *LANSAs for i User Guide* for more details about "next function" processing.

*PRO, which is the default value, indicates that the "next function" field should appear only when the process to which this function belongs has a menu selection style of "FUNCTION". If the process menu selection style is "NUMBER" or "CURSOR" then the next function field should not appear.

*YES indicates that the next function field should appear regardless of what menu selection style is being used by the process to which this function belongs.

*NO indicates that the next function field should not appear regardless of what menu selection style is being used by the process to which this function belongs.

Note: the SHOW_NEXT parameter is ignored in SAA/CUA applications.

Portability Considerations This feature is not known to Visual LANSAs and will be ignored, with no known effect to the application, if used in

Visual LANSAs code.

TEXT

Allows the specification of up to 50 "**text strings**" that are to appear on the screen panel or report. Each text string specified is restricted to a maximum length of 20 characters.

When a text string is specified it should be followed by a row/line number and a column/position number that indicates where it should appear on the screen panel or report.

For example:

```
TEXT(('ACME' 6 2)('ENGINEERING' 7 2))
```

specifies 2 text strings to appear at line 6, position 2 and line 7, position 2 respectively.

Portability Considerations In Visual LANSAs this parameter should only be edited using the screen or report painter which will replace any text with a text map. DO NOT enter text using the command prompt or free format editor as it will not pass the full function checker if checked in to LANSAs for i.

All Platforms

The text map is used by the screen or report design facilities to store the details of all the text strings associated with the screen panel or report lines.

Once a screen or report layout has been "painted" and saved, all text details from the layout are stored in a "text map". The text map is then subsequently changed by using the "painter" again.

The presence of a text map is indicated by a TEXT parameter that looks like this example:

```
TEXT((*TMAPnnn 1 1))
```

where "nnn" is a unique number (within this function) that identifies the stored text map.

Some very important things about "text maps" and *TMAPnnn identifiers that you must know are:

- Never specify *TMAPnnn identifiers of your own or change *TMAPnnn identifiers to other values. Leave the assignment and management of

*TMAPnnn identifiers to the screen and report design facilities.

- When copying a command that has an *TMAPnnn identifier, remove the *TMAPnnn references (ie: the whole TEXT parameter) from the copied command. If you fail to do this, then the full function checker will detect the duplicated use of *TMAPnnn identifiers, and issue a fatal error message before any loss occurs.
- Never remove an *TMAPnnn identifier from a command. If this is done then the associated text map may be deleted, or reused in another command, during a full function check or compilation. Loss of text details is likely to result.
- Never "comment out" a command that contains a valid *TMAPnnn identifier. This is just another variation of the preceding warning and it runs the same risks of loss or reuse of text.
- Never specify *TMAPnnn values in an Application Template. In the template context *TMAPnnn values have no meaning. Use the "text string" format in commands used in, and initially generated by, Application Templates.

CURSOR_LOC

Specifies any user controlled cursor positioning that is required. The CURSOR_LOC parameter must always contain 2 values, which may take any of the following forms:

*NONE / *NONE: which are the default values indicate that no user controlled cursor positioning is required. Normal LANSAs cursor control is to be used. When a screen is displayed the cursor will be positioned to either the first input capable field or the first field in error.

*ATFIELD / Field name: specifies that the cursor should be positioned to the named field. If the named field is not on the display or a field error exists, normal LANSAs cursor control will be used. Otherwise the cursor will be positioned to the nominated field.

Row value / Column value: specifies that the "values" nominated indicate the row and column number at which the cursor is to be positioned. The "values" nominated may be an alphanumeric literal (e.g.: 15) or the name of a field that contains the value (e.g.: #ROW). In all cases the value must be numeric. If the row or column values are invalid or a field error exists, normal LANSAs cursor control will be used. Otherwise the cursor will be positioned at the row and column specified.

When the row and column option is used **and** the row and column values are

specified as fields (rather than numeric literals), the row and column number that the cursor was at when the command completed execution will be returned in them.

Note: The CURSOR_LOC does not behave in the same way on Windows as on IBM i. On a Windows platform the value retrieved is the first position of the field the cursor is currently in.

The feature is a useful way of retrieving the location of the screen cursor at the time the command completed execution. In cases where you wish to retrieve the cursor location, but do not want to specify it before output to the screen, use coding like this:

```
CHANGE FIELD(#ROW #COL) TO(0)
REQUEST FIELDS(#FIELD1 .. #FIELD10) CURSOR_LOC(#ROW #COL)
```

When the REQUEST command is executed #ROW and #COL are both zero, which is an invalid cursor location. In such cases normal LANSAs cursor control is resumed and the user positioning request is ignored. However, after completion of the command fields #ROW and #COL will contain the location of the cursor at the time the REQUEST command completed execution.

STD_HEAD

Specifies whether or not the standard LANSAs design for the screen heading lines (lines 1 and 2) should be used.

*DFT, which is the default value, indicates that the system default value for the STD_HEAD parameter should be used. The system default value is stored in the LANSAs system definition block. Refer to [The System Definition Data Areas](#) in the *LANSAs for i User Guide* for details of the system definition block and how to change it.

*YES indicates that the standard LANSAs screen heading lines should be used. When this option is used lines 1 and 2 of the display are not available for the positioning of user fields.

*NO indicates that the standard LANSAs screen heading lines should not be used. In this case lines 1 and 2 of the display can be used to position user fields.

OPTIONS

Specifies special display options for this screen panel.

*NONE, which is the default value, indicates that there are no special display options for this screen panel.

Otherwise, specify one or more of the following:

***NOREAD** indicates that the details being displayed are not to be read back from the screen. Thus the details are presented to the user, but cannot ever be read back into the program. Additionally, the program does not stop at the command and wait for a user interaction. The stop and wait event will only occur when a subsequent DISPLAY or REQUEST command is executed that does not use the *NOREAD option.

***OVERLAY** indicates that the screen panel should overlay whatever details are already on the screen. Details already on the screen will become protected and can no longer be read from the device, but they will be visible to the user.

When ***OVERLAY** is used, the default for the STD_HEAD parameter is ***NO**. Therefore, unless STD_HEAD(*YES) is coded, the screen heading lines will not be displayed when using OPTIONS(*OVERLAY). Note that when a "standard heading" (*YES) is sent to the screen it causes the entire screen to be cleared. If STD_HEAD(*NO) is used it has no effect upon standard headings already on the screen from previous commands.

If either the ***NOREAD** or ***OVERLAY** options are used, then the complete screen details must fit on one screen panel.

Note: These display options have been provided to allow emulation of IBM i 3GL programs, and will not be portable to other platforms. They are not supported by the current GUI or by LANSAs for the Web. Use of these options is therefore not recommended.

Portability Not supported and should not be used in portable applications.
Considerations If used in Visual LANSAs code, a Full Function Check fatal error will be issued.

IGCCNV_KEY

Controls the appearance of the text "Fnn=XXXXXX" in the function key area, of the function key assigned to support IGC conversion.

This parameter is ignored if the language under which this function is being compiled does not have the "IGCCNV required" flag enabled, or if this function uses the *NOIGCCNV options keyword (refer to the FUNCTION command).

Also note that this parameter only controls the appearance of the text "Fnn=XXXXXX" in the function key area. It does not control the enablement of the IGCCNV DDS keyword in the display file associated with this function. This is controlled by the setting of the "IGCCNV required" flag and the use of the *NOIGCCNV option.

*AUTO, which is the default value, indicates that appearance of the function key text should be determined automatically. The automatic rules used to determine whether or not to show the function key text are:

- If there are no fields with keyboard shift J, E or O involved, the text will not appear (ignore all following rules).
- For a REQUEST command the text will always appear.
- For DISPLAY or POP_UP commands, the current "mode" is tested. If the mode is "change" (ie: fields on the screen are input capable), the text will appear. For all other modes the text will not appear.

Other allowable values for this parameter are *YES, indicating that the text should always appear, or, *NO indicating that the text should never appear.

The final option allows the nomination of a condition previously defined by a DEF_COND command. If the condition is true the text should appear. If the condition is false, the text should not appear.

Portability Will be ignored with no known effect to the application, if
Considerations used in Visual LANSA code.

7.80.2 REQUEST Comments / Warnings

The REQUEST command is **not** a "mode sensitive" command. For details of mode sensitive command processing, refer to [Screen Modes and Mode Sensitive Commands](#). All fields in the FIELDS parameter will be input capable unless they have the specific attributes *NOCHG or *OUTPUT.

The following table indicates all combinations of the DESIGN and IDENTIFY parameters and what values actually result when any of the default values are used:

Specified: DESIGN	Specified: IDENTIFY	LANS A Uses: DESIGN	LANS A Uses: IDENTIFY
*IDENTIFY	*DESIGN	*DOWN	*LABEL
*IDENTIFY	*COLHDG	*ACROSS	*COLHDG
*IDENTIFY	*LABEL	*DOWN	*LABEL
*IDENTIFY	*DESC	*DOWN	*DESC
*IDENTIFY	*NOID	*ACROSS	*NOID
*DOWN	*DESIGN	*DOWN	*LABEL
*DOWN	*COLHDG	*DOWN	*COLHDG
*DOWN	*LABEL	*DOWN	*LABEL
*DOWN	*DESC	*DOWN	*DESC
*DOWN	*NOID	*DOWN	*NOID
*ACROSS	*DESIGN	*ACROSS	*COLHDG
*ACROSS	*COLHDG	*ACROSS	*COLHDG
*ACROSS	*LABEL	*ACROSS	*LABEL
*ACROSS	*DESC	*ACROSS	*DESC
*ACROSS	*NOID	*ACROSS	*NOID

The following table indicates all combinations of the DESIGN and IDENTIFY parameters and what values result when the *DESIGN default is used in the associated DOWN_SEP or ACROSS_SEP parameters:

Specified: DESIGN	Specified: IDENTIFY	*DESIGN Specified: DOWN_SEP	*DESIGN Specified: ACROSS_SEP
*IDENTIFY	*DESIGN	1	1
*IDENTIFY	*COLHDG	5	1
*IDENTIFY	*LABEL	1	1
*IDENTIFY	*DESC	1	1
*IDENTIFY	*NOID	1	1
*DOWN	*DESIGN	1	1
*DOWN	*COLHDG	5	1
*DOWN	*LABEL	1	1
*DOWN	*DESC	1	1
*DOWN	*NOID	1	1
*ACROSS	*DESIGN	5	1
*ACROSS	*COLHDG	5	1
*ACROSS	*LABEL	1	1
*ACROSS	*DESC	1	1
*ACROSS	*NOID	1	1

- In some cases all the fields specified in the FIELDS parameter will not fit on one screen. In this case a second, third, fourth, etc screen is automatically designed as required.
- In terms of the RDML program they can be treated like one "long" screen. LANSA will automatically process the screens one after another until they have all been processed. When all screens have been processed the next RDML command is executed. So when you use the REQUEST command you

may in fact be requesting that 2 or 3 or more screens be input one after another.

- This facility is a feature of the automatic design procedures. If you are coding the RDML program yourself it may be advisable in some circumstances to "split up" the REQUEST command into multiple REQUEST commands that have only one screen format each.
- If you want the name of the field in which the CURSOR was located when Enter or any other AID was pressed, to be returned to your function, then refer to a field named #CURLOC\$FN within your function.
#CURLOC\$FN (alpha, 10) will contain the name of the field.

7.80.3 REQUEST Examples

Example 1: Input fields #ORDNUM, #CUSTNUM and #DATEDUE from the workstation:

```
REQUEST FIELDS(#ORDNUM #CUSTNUM #DATEDUE)
```

or, identically:

```
GROUP_BY NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #  
REQUEST FIELDS(#ORDERHEAD)
```

both use default values for all parameters and field attributes and thus would cause a screen something like this to be designed automatically:

```
Order number : _____  
Customer no  : _____  
Date due    : _____
```

Example 2: Modify the previous example to design the screen across ways and use column headings to identify the fields:

```
GROUP_BY NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #  
REQUEST FIELDS(#ORDERHEAD) DESIGN(*ACROSS) IDENTIFY(*C
```

which would cause a screen something like this to be designed automatically:

```
Company   Order   Date  
Order     Customer Order  
Number    Number   Due  
_____  
_____
```

Example 3: Request #ORDNUM #CUSTNUM and #DATEDUE and also specify specific positions and identification methods as field attributes.

For details of field attributes, refer to [Field Attributes and their Use](#).

When specific positions for a field are nominated the automatic design facility is effectively "disabled".

```
GROUP_BY NAME(#ORDERHEAD) FIELDS((#ORDNUM *COLHDG *I
```

REQUEST FIELDS(#ORDERHEAD) TEXT(('--DATE--' 6 37))

which would cause a screen something like this to be designed:

Company Customer no : _____
Order
Number
_____ --DATE--

Note that the manual specification of row and column numbers and "text" is not required. The screen design facility can be used to modify an "automatic" design much more quickly and easily. Refer to the *LANSA for i User Guide* for details of how to use the screen design facility.

After the screen design facility has been used on a REQUEST command the associated FIELDS parameter (in the REQUEST or GROUP_BY command) will be automatically re-written with the required row, column and method of identification attributes.

In addition the TEXT parameter of the REQUEST command will also be automatically re-written.

Example 4: Use an Expandable Group expression and redesign the layout using the screen design facility:

```
GROUP_BY NAME(#XG_ORDHDG) FIELDS(#ORDNUM #CUSTNUM #  
REQUEST FIELDS(#XG_ORDHDG) DESIGN(*ACROSS) IDENTIFY(*C
```

The screen designed automatically would look like:

Company Order Date
Order Customer Order
Number Number Due

If the layout is changed using the screen design facility to look like this:

Company Order

Order Customer
Number Number
_____ _____ Date Order Due _____

The REQUEST command FIELDS parameter will be expanded as follows:

REQUEST FIELDS((#ORDNUM *L2 *P3) (#CUSTNUM *L2 *P13) (#DA'

7.81 RETURN

The RETURN command is used to cause an executing subroutine or function to end and return control to the calling function or process. The calling function may be the function "mainline", another subroutine within the function or even another process altogether.

When a RETURN command is executed in the "mainline" of a function the current function (and the process to which it belongs) is ended and control is returned to the process or function that invoked it. See the CALL command for more details of how a process or function can be invoked from another function.

When a RETURN command is executed within a subroutine (see the SUBROUTINE command) control is returned to the command following the EXECUTE command that caused the subroutine to be invoked. The EXECUTE command may have been in either another subroutine or in the "mainline" of the function.

Also See

[7.81.1 RETURN Parameters](#)

[7.81.2 RETURN Examples](#)

RETURN ----- no parameters -----

|

7.81.1 RETURN Parameters

The RETURN command has no parameters.

7.81.2 RETURN Examples

In this example, a subroutine uses the RETURN command rather than a complex IF-ELSE structure:

```
SUBROUTINE NAME(MATHS) PARMS(#F1 #OP #F2 #RS)

DEFINE    FIELD(#F1) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE    FIELD(#OP) TYPE(*CHAR) LENGTH(1)
DEFINE    FIELD(#F2) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE    FIELD(#RS) TYPE(*DEC) LENGTH(15) DECIMALS(5)

IF        COND('#OP = "+"')
CHANGE    FIELD(#RS) TO('#F1 + #F2')
RETURN
ENDIF

IF        COND('#OP = "-"')
CHANGE    FIELD(#RS) TO('#F1 - #F2')
RETURN
ENDIF

IF        COND('#OP = "*"')
CHANGE    FIELD(#RS) TO('#F1 * #F2')
RETURN
ENDIF

IF        COND('#OP = "/"')
CHANGE    FIELD(#RS) TO('#F1 / #F2')
RETURN
ENDIF

MESSAGE   MSGTXT('Operation specified is not +,-,* or /')

ENDROUTINE
```

7.82 ROLLBACK

The rollback command is used to cause an IBM i operating system "rollback" operation to be issued. A rollback operation "rolls back" (ie: removes) all uncommitted changes from the database.

Refer to the appropriate IBM supplied manual for details of IBM i commitment control processing before attempting to use this command.

It is also advisable to read [Commitment Control](#) in the *LANSA for i User Guide*.

Portability If using Visual LANSA, refer to [Commitment Control](#) in the
Considerations *LANSA Application Design Guide*.

Also See

[7.82.1 ROLLBACK Parameters](#)

[7.82.2 ROLLBACK Examples](#)

ROLLBACK ----- no parameters -----
-|

7.82.1 ROLLBACK Parameters

The ROLLBACK command has no parameters.

7.82.2 ROLLBACK Examples

Requesting the user to input details of an order. Write the order header and all associated lines to the database. Ask the user to confirm that the order should be kept. If the reply is not YES, remove the order from the database.

```
GROUP_BY  NAME(#ORDERHEAD) FIELDS(#ORDNUM #CUSTNUM #
DEF_LIST  NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QUA
DEFINE    FIELD(#CONFIRM) TYPE(*CHAR) LENGTH(3) LABEL('Conf
```

```
SET_MODE  TO(*ADD)
INZ_LIST  NAMED(#ORDERLINE) NUM_ENTRYYS(20)
REQUEST   FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)
```

```
INSERT    FIELDS(#ORDERHEAD) TO_FILE(ORDHDR)
SELECTLIST NAMED(#ORDERLINE) GET_ENTRYYS(*NOTNULL)
INSERT    FIELDS(#ORDERLINE) TO_FILE(ORDLIN)
ENDSELECT
```

```
REQUEST   FIELDS((#CONFIRM)(#ORDNUM *NOCHG))
```

```
IF        COND('#CONFIRM = YES')
COMMIT
MESSAGE   MSGTXT('Order has been committed to the database')
ELSE
ROLLBACK
MESSAGE   MSGTXT('Order has been removed from the database')
ENDIF
```

If the function were to fail when writing the 4th order line (say), then an automatic rollback would be issued. This would cause the order header and any order lines already created to be rolled back from the file.

7.83 SELECT

The SELECT command is used in conjunction with the ENDSELECT command to form a "loop" to process one or more records from a file that match certain criteria.

For example, the SELECT / ENDSELECT loop:

```
--->SELECT FIELDS(#ORDLIN #PRODUCT #QUANTITY)
|   FROM_FILE(ORDLIN) WITH_KEY(#ORDER)
|
|   DISPLAY FIELDS(#ORDER #ORDLIN #PRODUCT #QUANTITY)
|
----ENDSELECT
```

Forms a loop to read all records from file ORDLIN that have an order number matching the value in field #ORDER.

Each time a record is read the DISPLAY command, which is within the SELECT / ENDSELECT loop will display details of the record just read.

The SELECT command is probably the most flexible command in the LANSA RDML and some experience with it is required before the full power can be utilized. Some of the types of database processing supported by it include:

- Entry sequence processing
- Full key processing
- Partial key processing
- Generic key processing
- Execution time modification of the number of keys to be used
- Conditional selection of records
- Forwards or backwards processing of selected records
- Start at (or near) a key then process backwards or forwards

In addition, the SELECT command can be used in conjunction with the IBM i operating system command OPNQRYP (Open Query File). This extends the power of the SELECT command to include:

- Execution time modification of record selection criteria
- Execution time modification of the order records are processed

- Field content searching
- Field substringing during selection comparisons
- Searching without regard to the case (upper or lower) of fields

For more details of how to use the IBM i operating system command OPNQRYF refer to the OPEN command in this guide first.

SELECT loop logic that should be avoided.

When fields A, B and C are selected in a SELECT loop like this:

```
SELECT FIELDS(#A #B #C)
FROM_FILE(...)
WHERE(.....)
.....
.....
ENDSELECT
```

they have a predictable and consistent value within the loop across all platforms. These fields do not have a predictable and consistent value outside the loop. So this:

```
SELECT FIELDS(#A #B #C) FROM_FILE(...)
.....
IF COND(#A < 35.5)
.....
ENDIF
.....
ENDSELECT
```

is a predictable piece of logic, while:

```
SELECT FIELDS(#A #B #C)
FROM_FILE(...)
WHERE(.....)
.....
.....
ENDSELECT
IF COND(#A < 35.5)
```

.....
ENDIF

in any form or variation, is an unpredictable piece of logic.

The value of A (B and C), in terms of data read from the selection table, after exit from the SELECT loop, are actually defined as "not defined". This means that their values at the termination of a SELECT / ENDSELECT loop are not predictable or consistent across platforms.

Portability Considerations Refer to parameters [FROM_FILE](#) , [GENERIC](#) , [LOCK](#) and [OPTIONS](#) .

Also See

[7.83.1 SELECT Parameters](#)

[7.83.2 SELECT Comments / Warnings](#)

[7.83.3 SELECT Examples](#)

Required

SELECT ----- *FIELDS* ----- *field name field attributes* -
-->

```
|          |          ||  
|          --- 7 max ----- |  
|*ALL          |  
|*ALL_REAL          |  
|*ALL_VIRT          |  
|*INCLUDING          |  
|*EXCLUDING          |  
|expandable group          |  
|          |  
|----- 1000 max for RDMLX---|  
|----- 100 max for RDML ----|
```

>-- *FROM_FILE* ---- *file name . *FIRST* -----
>
library name

```

-----
                                Optional
>-- WHERE ----- 'condition' ----->

>-- WITH_KEY ----- key field values ----->
      expandable group expression

>-- NBR_KEYS ----- *WITHKEY -----
->
      *COMPUTE
      numeric field name

>-- GENERIC ----- *NO ----->
      *YES

>-- IO_STATUS ----- *STATUS ----->
      field name

>-- IO_ERROR ----- *ABORT -----
>
      *NEXT
      *RETURN
      label

>-- VAL_ERROR ----- *LASTDIS -----
>
      *NEXT
      *RETURN
      label

>-- END_FILE ----- *NEXT ----->
      *RETURN
      label

>-- ISSUE_MSG ----- *NO ----->
      *YES

>-- LOCK ----- *NO ----->
      *YES

```

-> >-- RETURN_RRN --- *NONE -----

field name

>-- OPTIONS ----- up to 5 options allowed -----|

**BACKWARDS*

**STARTKEY*

**ENDWHERE*

**ENDWHERESQL*

**BLOCKnnn*

7.83.1 SELECT Parameters

END_FILE

FIELDS

FROM_FILE

GENERIC

IO_ERROR

IO_STATUS

ISSUE_MSG

LOCK

NBR_KEYS

OPTIONS

RETURN_RRN

VAL_ERROR

WHERE

WITH_KEY

FIELDS

Specifies either the field(s) that are to be selected from the record in the file or the name of a group that specifies the field(s) to be selected. Alternatively, an expandable group expression can be entered in this parameter.

The following special values can be used:

- *ALL specifies that all fields from the currently active file be selected.
- *ALL_REAL specifies that all real fields from the currently active file be selected.
- *ALL_VIRT specifies that all virtual fields from the currently active file be selected.
- *EXCLUDING specifies that fields following this special value must be excluded from the field list.
- *INCLUDING specifies that fields following this special value must be included in the field list. This special value is only required after an *EXCLUDING entry has caused the field list to be in exclusion mode.

Note: When all fields are selected from a logical file maintained by OTHER, all the fields from the based-on physical file are included in

the field list.

It is strongly recommended that the special values *ALL, *ALL_REAL or *ALL_VIRT in parameter FIELDS be used sparingly and only when strictly required. Selecting fields which are not needed invalidates cross-reference details (shows fields which are not used in the function) and increases the Crude Entity Complexity Rating of the function pointlessly.

Note that when BLOB or CLOB data is retrieved, it is either *SQLNULL or a filename. If a filename, the data from the database file has been copied into the file.

Warning: It is time-consuming to retrieve BLOB or CLOB fields from a file.

Recommended Database Design When Using BLOB and CLOB Fields

The recommended design when using BLOB and CLOB fields is to put them in a separate file from the rest of the fields using the same key as the main file. This forces programmers to do separate IOs to access the BLOB and CLOB data, thus reducing impact on database performance from indiscriminate use of this data. It is also the most portable design ensuring that the non-BLOB and non-CLOB data can be quickly accessed at all times.

FROM_FILE

Refer to [Specifying File Names in I/O commands](#) .

WHERE

Refer to [Specifying Conditions and Expressions](#) and [Specifying WHERE Parameter in I/O Commands](#).

After a SELECT/ENDSELECT loop utilizing a where condition, the contents of the fields are unpredictable. The records matching the where condition should only be processed within the SELECT/ENDSELECT loop.

WITH_KEY

Refer to [Specifying File Key Lists in I/O Commands](#) .

NBR_KEYS

This parameter can be used in conjunction with the WITH_KEY parameter to vary the number of key fields that are actually used to retrieve records at execution time.

*WITHKEY, which is the default value, specifies that the number of key fields will always match the number specified in the WITH_KEY parameter and the

value will not be changed at execution time.

*COMPUTE can also be specified for this parameter. This specifies that the number of keys should be determined by examining the **contents** of the fields nominated in the WITH_KEY parameter at execution time.

The logic used to determine the number of keys works like this:

Set <n> to number of fields specified in WITH_KEY parameter.

Do while n is greater than zero and keyfield(n) is *NULL or *SQLNULL.

Subtract 1 from n.

Endwhile.

Set <number of keys> to <n>.

For a definition of the *NULL value for each of the field types, refer to [7.9.1 CHANGE Parameters](#).

If you want to **vary the number of key fields** by **direct RDML logic** specify **the name of a numeric field** for this parameter. The field you name should contain the number of keys value at execution time. The field specified must be defined in this function or in the LANSAs data dictionary and must be numeric.

At execution time the value contained in the NBR_KEYS field must be not less than zero and not greater than the number of key fields specified in the WITH_KEY parameter.

When a SELECT command is executed with the NBR_KEYS field set to zero the entire WITH_KEY parameter is effectively ignored for selection purposes.

Refer to the examples following for more information.

GENERIC

Specifies whether or not generic searching is required. Generic searching is different to full or partial key searching because only the non-blank or non-zero portion of the key value is used when comparing the search key with the file key.

GENERIC is ignored for Date, Time, Datetime, Integer, and Float.

When using generic searching on an alphanumeric field only the leftmost non-blank portion of the search field is compared with the file key (i.e., trailing blanks are ignored for comparative purposes).

When using generic searching on a numeric field only the leftmost non-zero portion of the search field is compared with the file key (ie: trailing zeros are ignored for comparative purposes). Also, generic numeric field comparisons are done as if both the search key and the file key are positive numbers, regardless

of what they actually are.

Note that these generic search rules mean that a blank alphanumeric search key or a zero (0) numeric key will match every record selected.

For example, if a file was keyed by a name field and it contained the following values:

SM

SMIT

SMITH

SMITHS

SMITHY

SMYTHE

then the SELECT statement:

```
SELECT WITH_KEY('SM')
```

would only select the first record in the file because it is the only record that matches the full key value 'SM'. If however, the SELECT statement was changed to:

```
SELECT WITH_KEY('SM') GENERIC(*YES)
```

then all the records in the file would be selected because only the non-blank portion of the key value specified is compared with the file key.

*NO, which is the default value, indicates that generic searching is not required.

*YES indicates that generic searching is to be performed. When generic searching is used it is only actually performed on the last key that was supplied at execution time. Other (previous) keys specified in the WITH_KEY parameter must exactly match the values in the file. They are not generically compared

with the data in the file.

For instance, imagine a name and address file that is keyed by state, post/zip code and name. The command:

```
SELECT WITH_KEY('NSW' 2000 'SM') NBR_KEYS(3) GENERIC(*YES)
```

would select all names in NSW, with post code 2000 whose names start with SM. This is an example of generic searching on an alphanumeric field. Trailing blanks are ignored when comparing the search key with the data read from the file. Also note that only the last key ('SM') is actually generically compared with the data on the file. The other keys , 'NSW' and 2000 must exactly match data read from the file. If, at execution time the command was dynamically modified to use 2 keys, like this:

```
SELECT WITH_KEY('NSW' 2000 'SM') NBR_KEYS(2) GENERIC(*YES)
```

then it would select all names in NSW with a post code that starts with 2 (ie: 2000 to 2999). This is an example of generic searching on a numeric field where trailing zeroes (0's) are ignored.

Portability Considerations When native I/O is used, there is an implied *ENDWHERE when a key is encountered that does not match the search key generically. You must test the application to confirm that it is functioning as required.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

For values, refer to [I/O Return Codes](#) .

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command. The purpose of *NEXT is to permit you to handle error messages in the RDML, and then ABORT, rather than use the default ABORT. (It is possible for processing to continue for LANSAs for i and Visual LANSAs, but this is NOT a recommended way to use LANSAs.)

ER returned from a database operation is a fatal error and LANSAs does not expect processing to continue. The IO Module is reset and further IO will be as if no previous IO on that file had occurred. Thus you must not make any presumptions as to the state of the file. For example, the last record read will not be set. A special case of an IO_ERROR is when a trigger function is coded to return ER in TRIG_RETC. The above description applies to this case as well. Therefore, LANSAs recommends that you do NOT use a return code of ER from a trigger function to cause anything but an ABORT or EXIT to occur before any further IO is performed.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

VAL_ERROR

Specifies the action to be taken if a validation error was detected by the command.

A validation error occurs when information that is to be added, updated or deleted from the file does not pass the FILE or DICTIONARY level validation checks associated with fields in the file.

If the default value *LASTDIS is used control will be passed back to the last display screen used. The field(s) that failed the associated validation checks will be displayed in reverse image and the cursor positioned to the first field in error on the screen.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).

When using *LASTDIS the "Last Display" must be at the same level as the database command (INSERT, UPDATE, DELETE, FETCH and SELECT). If they are at different levels e.g. the database command is specified in a SUBROUTINE, but the "Last Display" is a caller routine or the mainline, the function will abort with the appropriate error message(s).

The same does NOT apply to the use of event routines and method routines in Visual LANSA. In these cases, control will be returned to the calling routine. The fields will display in error with messages returned to the first status bar encountered in the parent chain of forms, or if none exist, the first form with a status bar encountered in the execution stack (for example, a reusable part that inherits from PRIM_OBJT).

END_FILE

Specifies what is to happen when the "end of the file" is reached. Note that the "end of the file" means the last record that matches the selection criteria has been processed, not necessarily the last record in the file has been processed.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

Values may not be as expected on exit from a select loop due to an extra record

being read to determine when to terminate the select loop.

ISSUE_MSG

Specifies whether an "end of file" message is to be automatically issued or not. The default value is *NO which indicates that no message should be issued.

The only other allowable value is *YES which indicates that a message should be automatically issued. The message will appear on line 22/24 of the next screen format presented to the user or on the job log of a batch job.

LOCK

Specifies whether or not the record should be locked when it is read.

*NO, which is the default value, indicates that the record should not be locked.

*YES indicates the record should be locked. It is the responsibility of the user to ensure that the record is released at some future time.

Note: LOCK(*YES) performs a record level lock. It may exhibit intra and inter operating system behavioral variations (e.g. commitment control locking multiple records; default wait times). User's are advised to investigate the development of proper and complete "user object" locking protocol by using the LOCK_OBJECT Built-In Function.

Portability Not supported and should not be used in portable applications.
Considerations A build warning will be generated when used in Visual LANSAs.

RETURN_RRN

Specifies the name of a field in which the relative record number of the record just selected should be returned.

Any field nominated in this parameter must be defined within the function or the LANSAs data dictionary and must be numeric.

Note: The value returned by this parameter, when OPTIONS(*BLOCKnnn) is used, is largely useless, as it represents the relative record number of the last record in the block of records just read, which may not be the number of the record currently being processed by the SELECT/ENDSELECT loop.

For further reference refer to [Load Other File](#) in the *Visual LANSAs Developers Guide*.

OPTIONS

Specifies from 1 to 4 special options that can be used when processing records

from the file. Allowable special options are:

***BACKWARDS:** Indicates that the records should be processed in reverse order to that which would normally be used. Normally records are read in the order of the key specified in the FROM_FILE parameter.

When the *BACKWARDS option is used the records are read in reverse order. Backwards processing by sequential, full or partial key is supported (even though it is not supported by some other high level languages such as RPG). Generic key processing backwards is also supported (but may be hard to effectively implement).

***STARTKEY:** Indicates that the key(s) nominated in the WITH_KEY parameter should only be used to establish the start position for the first read operation.

The first record read will be the first one that has a key greater than or equal to the key value(s) nominated in the WITH_KEY parameter. All subsequent records are then processed with no regard to the WITH_KEY values. In this situation the SELECT loop normally has to be terminated by program logic or by using the special option *ENDWHERE.

***ENDWHERE:** Specifies that if the condition specified in the WHERE parameter is found to be false (that is, not true) then the SELECT loop should terminate. Control is then passed to the position in the program nominated by the END_FILE parameter.

Normally a SELECT loop only terminates when all records that could match the selection criteria have been read and examined for possible selection and processing by the SELECT loop. When the *ENDWHERE option is used, the first time a record is read (or some other condition occurs) that causes the WHERE condition to be false, the SELECT loop is terminated.

Portability Considerations Use of the SELECT options *STARTKEY and *ENDWHERE are not recommended for portable applications as they may have performance implications when using SQL requests.
The *STARTKEY option emulates the positioning of a "file cursor".
The *ENDWHERE option tests the condition inside the select loop and is not placed in the WHERE clause when using SQL.

***ENDWHERE SQL:** allows you to handle SELECT commands in the most appropriate manner according to the WHERE condition.

SQL (using ODBC) doesn't handle table operations the same way as native I/O on IBM i . To enhance the performance of the SELECT command for different tables (SQL and native I/O), this value supersedes *ENDWHERE.

This new value is applied by selecting the *force *ENDWHERE*SQL option in the Partition definition tab. This partition option signals to LANSAs that ALL *ENDWHERE options in SELECT commands are to be interpreted as though *ENDWHERESQL had been coded. As code is updated, or new code is written, it is recommended that the SELECT commands are changed to use this new option where it is appropriate.

Portability Considerations This value flags the runtime to interpret SELECT operation differently:

- If the code is generated to use SQL, then this *ENDWHERESQL option will effectively be ignored and the WHERE condition will be placed in the WHERE clause of the SQL request (when all real columns are specified).
- If the code is generated to use native I/O access, then the *ENDWHERESQL option will be interpreted to be the same as the *ENDWHERE option.

* BLOCKnnn: Specifies that the records selected from the file are to be read in blocks to reduce the number of real database I/O operations being performed.

Used properly, this option can substantially improve the performance of a SELECT/ENDSELECT loop.

However, LANSAs and IBM i conditions apply to using it:

- The 'nnn' component of the *BLOCKnnn parameter value specifies the number of records read in each block. The allowable values for 'nnn' are 010, 020, 030, 040, 050, 060, 070, 080, 090, 100, 150, 200, 250, 300, 400 or 500. The IBM recommended value is as many records as will fit into a 32K block.
- The use of *BLOCKnnn implies the use of *DBOPTIMISE, regardless of whether or not *DBOPTIMISE is actually specified in a FUNCTION command.
- The SELECT command must not have a WITH_KEY parameter. The use of the WITH_KEY parameter causes file cursor positioning operations that disable the blocking logic.

- No other I/O operations must be performed on the file specified in the FROM_FILE parameter anywhere else in the function.
- The SELECT/ENDSELECT loop must be executed once and only once in the programs invocation. Subsequent attempts to (re)execute the SELECT/ENDSELECT loop will cause unpredictable results because the file cursor will not be (re)positioned to the start of the file. If this feature is required, CLOSE the file before attempting to (re)execute the SELECT/ENDSELECT loop.
- The file must not be left open by option *KEEPOPEN or have been opened (or left open) by some other function / program, including this one. If the file is already open the current location of the file cursor is unpredictable. If in doubt, use the CLOSE command to close the file first.
- The relative record number returned by the RETURN_RRN parameter is meaningless when OPTIONS(*BLOCKnnn) is used as it represents the number of the last record read in the current block, which may not be the record being processed by the SELECT/ENDSELECT loop.

Portability Not supported and should not be used in portable applications.

Considerations A Full Function Check fatal error will be issued when used in Visual LANSA. *BLOCKnnn options are ignored with no known effect to the application.

7.83.2 SELECT Comments / Warnings

SQL does not handle all of its table operations in the same manner as file operations on the IBM i. Here are some important points which you should be aware of:

- The SQL based SELECT operation may select all the matching rows at the time it is **first executed** (into a temporary table) and then proceed to process the selected set of rows (one by one).

This style of processing **may** cause functional changes between IBM i and Visual LANSA applications where a SELECT loop actually inserts or updates rows, as it goes, so that they become part of the set of "selectable" rows.

Under IBM i such rows would be processed by the SELECT loop. Under Visual LANSA they may not, because they were not part of the initially selected set.

This style of processing would be **quite strange** under IBM i because it runs a very real risk of infinite loops, but this processing difference should be noted and you should avoid this style of processing.

- DO NOT break SELECT loops with GOTO commands as this may leave the SQL cursor open. You should use the LEAVE RDML command to exit SELECT loops instead.
- DO NOT under any circumstances branch into the middle of a SELECT loop. This is an unnatural coding technique that will produce unpredictable results on any platform.
- For similar reasons to the previous points, changing the value of selection criteria within a SELECT loop may produce platform variant results. Consider this SELECT loop where SALARY is a column in the SQL table and REQSALARY is some sort of selection value:

```
SELECT FIELDS(...) FROM_FILE(...) WHERE('#SALARY <
    #REQSALARY')
.....
.....
.....
CHANGE #REQSALARY ('#REQSALARY * 1.1')
```

ENDSELECT

- This is not a well written piece of logic **and** it may produce differing results between platforms.
- Visual LANSA evaluates and selects according to the values at the time that the SELECT is first executed. Do **not** change the value of selection criteria after they have been established or unpredictable results may occur.
- The SQL-based SELECT operation with a WHERE condition involving only "real" fields in the named file selects only the matching set of rows from the file. This means that after read triggers **will not** be invoked for those rows not matching the WHERE condition.

This may be functionally different to RDML on the IBM i where all rows are read before testing the WHERE condition

If the WHERE condition involves any fields that are not "real" fields in the file then the processing will be identical under Visual LANSA and IBM i (RDML and RDMLX).

- Generic search where the search string contains the '%' character may not act the same as on the IBM i.

Refer to the SQL LIKE predicate in the appropriate SQL Reference manual.

- Use of the SELECT option *STARTKEY is not recommended as it may have performance implications when using SQL requests to emulate the positioning of a "file cursor".

7.83.3 SELECT Examples

Example 1: Select and print fields #ORDLIN, #PRODUCT, #QUANTITY and #PRICE from all records in an order lines file which have an order number matching that specified in field #ODRNUM.

```
SELECT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE) FROM_  
UPRINT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)  
ENDSELECT
```

or identically:

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU.  
  
SELECT  FIELDS(#ORDERLINE) FROM_FILE(ORDLIN) WITH_KEY(#(  
UPRINT  FIELDS(#ORDERLINE)  
ENDSELECT
```

Example 2: Select and print fields #ORDLIN, #PRODUCT, #QUANTITY and #PRICE from all records in an order lines file which have a #QUANTITY value greater than 10 or a #PRICE value less than 49.99

```
SELECT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE) FROM_  
UPRINT  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)  
ENDSELECT
```

or identically:

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU.  
  
SELECT  FIELDS(#ORDERLINE) FROM_FILE(ORDLIN) WHERE('('(#QU  
UPRINT  FIELDS(#ORDERLINE)  
ENDSELECT
```

Example 3: If a file called ACCOUNT contains the following fields and data:

Company (#COMP)	Division (#DIV)	Department (#DEPT)	Expenditure (#EXPEND)	Revenue (#REVNU)
01	1	ADM	400	576
"	"	MKT	678	56

"	"	SAL	123	6784
"	2	ADM	46	52
"	"	SAL	978	456
"	3	ACC	456	678
"	"	SAL	123	679
02	1	ACC	843	400
"	"	MKT	23	0
"	"	SAL	876	10
"	2	ACC	0	43

and the file is keyed by #COMP, #DIV and #DEPT then use the NBR_KEYS parameter of the SELECT command to create a very flexible browse function:

```
DEF_LIST NAME(#ACCOUNTS) FIELDS(#COMP #DIV #DEPT #EXPEI
DEFINE FIELD(#NBRKEYS) TYPE(*DEC) LENGTH(1) DECIMALS(0)
```

```
BEGIN_LOOP
```

```
CHANGE (#COMP #DIV #DEPT) *NULL
REQUEST FIELDS(#COMP #DIV #DEPT)
```

```
IF_NULL FIELD(#COMP #DIV #DEPT)
```

```
CHANGE #NBRKEYS 0
```

```
ELSE
```

```
IF_NULL FIELD(#DIV #DEPT)
```

```
CHANGE #NBRKEYS 1
```

```

ELSE
IF_NULL  FIELD(#DEPT)
CHANGE  #NBRKEYS 2
ELSE
CHANGE  #NBRKEYS 3
ENDIF
ENDIF
ENDIF
CLR_LIST NAMED(#ACCOUNTS)
SELECT  FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT) WITH_KEY
ADD_ENTRY TO_LIST(#ACCOUNTS)
ENDSELECT

DISPLAY  BROWSELIST(#ACCOUNTS)

END_LOOP

```

If the user does not input any values at the REQUEST command, then #NBRKEYS will contain 0 when the SELECT command is executed, so in effect the SELECT command that is being executed is:

```
SELECT  FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT)
```

which causes all the records in the file to be displayed.

If the user inputs a value for #COMP at the REQUEST command, then #NBRKEYS will contain 1 when the SELECT command is executed, so in effect the SELECT command that is being executed is:

```
SELECT  FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT) WITH_KEY(#
```

which causes all the records in the file that have the requested company number to be displayed.

If the user inputs a value for #COMP and a value for #DIV at the REQUEST command, then #NBRKEYS will contain 2 when the SELECT command is executed, so in effect the SELECT command that is being executed is:

```
SELECT  FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT) WITH_KEY(#
```

which causes all the records in the file that have the requested company number and division number to be displayed.

If the user inputs a value for #COMP, a value for #DIV and a value for #DEPT at the REQUEST command, then #NBRKEYS will contain 3 when the SELECT command is executed, so in effect the SELECT command that is being executed is:

```
SELECT FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT) WITH_KEY(#
```

which causes all the records in the file that have the requested company number, division number and department number to be displayed. For the data specified, only one record would ever be displayed in this case.

Example 4: Produce a functionally identical solution to example 3 by using the NBR_KEYS(*COMPUTE) parameter:

```
DEF_LIST NAME(#ACCOUNTS) FIELDS(#COMP #DIV #DEPT #EXPEI  
  
BEGIN_LOOP  
CHANGE (#COMP #DIV #DEPT) *NULL  
REQUEST FIELDS(#COMP #DIV #DEPT)  
CLR_LIST NAMED(#ACCOUNTS)  
SELECT FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT) WITH_KEY(  
ADD_ENTRY TO_LIST(#ACCOUNTS)  
ENDSELECT  
DISPLAY BROWSELIST(#ACCOUNTS)  
END_LOOP
```

Example 5: Select and print fields #ORDLIN, #PRODUCT, #QUANTITY and #PRICE from all records in an order lines file which have an order number matching that specified in field #ODRNUM. Print the information in reverse order (ie: highest line number first).

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU.  
  
SELECT FIELDS(#ORDERLINE) FROM_FILE(ORDLIN)  
WITH_KEY(#ORDNUM) OPTIONS(*BACKWARDS)  
UPRINT FIELDS(#ORDERLINE)  
ENDSELECT
```

Example 6: Use exactly the same logic as example 5, but ensure that no more than 3 lines are ever printed.

```
GROUP_BY NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #QU.
```

```

CHANGE FIELD(#COUNTER) TO(0)
SELECT FIELDS(#ORDERLINE) FROM_FILE(ORDLIN) WITH_KEY(#C
UPRINT FIELDS(#ORDERLINE)
CHANGE FIELD(#COUNTER) TO('#COUNTER + 1')
ENDSELECT

```

Example 7: Ask the user to input a customer name. Then select and display details of the first 10 names from a name and address file that are "closest" to the nominated name.

```

DEF_LIST NAME(#CUSTOMER) FIELDS(#NAME #CUSTNO #ADDR1 #

REQUEST FIELDS(#NAME)
CLR_LIST NAMED(#CUSTOMER)
SELECT FIELDS(#CUSTOMER) FROM_FILE(NAMES) WITH_KEY(#N
ADD_ENTRY TO_LIST(#CUSTOMER)
ENDSELECT
DISPLAY BROWSELIST(#CUSTOMER)

```

Example 8: Select all fields from the currently active version of file ORDLIN, perform diverse calculations involving all fields from the file and print the results for each selected record.

```

SELECT FIELDS(*ALL) FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM)
.....
.....
.....

UPRINT FIELDS(#RESULT1 #RESULT2 #RESULT3)

```

7.84 SELECTLIST

The SELECTLIST command is used in conjunction with the ENDSELECT command to form a loop to process all entries from a list that match certain criteria.

The list may be a **browse** list (used for displaying information at a workstation) or a **working** list (used to store information within a program).

For example, the SELECTLIST / ENDSELECT loop:

```
-----> SELECTLIST NAMED(#ORDERLINE) GET_ENTRY(*ALTERED)
| UPDATE   FIELDS(#ORDERLINE) IN_FILE(ORDLIN)
|         WITH_KEY(#ORDER)
----- ENDSELECT
```

forms a loop to process all entries in a list named #ORDERLINE that have been altered since they were added to it. Each changed entry is processed by the UPDATE command which reflects the change into file ORDLIN.

For more details of what lists are and list processing refer to the DEF_LIST (define list) command.

Also See

[7.84.1 SELECTLIST Parameters](#)

[7.84.2 SELECTLIST Examples](#)

Optional

```
SELECTLIST --- NAMED ----- *FIRST -----
----->
```

list name

```
>-- GET_ENTRY* --- *ALL ----->
```

```
*SELECT <-----
```

```
*ALTERED | Valid for
```

```
*NOTNULL | browse lists
```

```
*DISPLAY | only
```

```
*ADD |
```

**CHANGE* |
**DELETE* <-----

>--- *WHERE* ----- '*condition*' -----|

7.84.1 SELECTLIST Parameters

[GET_ENTRIES](#)

[NAMED](#)

[WHERE](#)

NAMED

Specifies the name of the list that is to be processed by the SELECTLIST / ENDSELECT loop.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which may be a browse or a working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

GET_ENTRIES

Specifies which entries from the list are to be processed by the SELECTLIST / ENDSELECT loop.

*ALL, which is the default value, indicates that all entries in the list should be processed by the loop. When the list being processed is a working list this is the parameter value that is valid.

*SELECT indicates that only entries that have a non-blank "select" field should be processed by the loop. A "select" field in a list has the field attribute *SELECT associated with it. For more details, refer to the [DEF_LIST](#) command. Also see how [field attributes](#) are used.

*ALTERED indicates that only entries that have been altered since they were added to the list by the ADD_ENTRY or INZ_LIST command should be processed by the loop.

*NOTNULL indicates that only entries that have a non "null" value in one or more of the fields should be processed by the loop. The "null" value is blanks for alphanumeric fields and zero for numeric fields.

*DISPLAY indicates that only entries that were added or updated in the list when the screen processing mode was *DISPLAY should be processed by the loop. For more details, refer to [Screen Modes and Mode Sensitive Commands](#).

*ADD indicates that only entries that were added or updated in the list when the screen processing mode was *ADD should be processed by the loop. For more details, refer to [Screen Modes and Mode Sensitive Commands](#).

*CHANGE indicates that only entries that were added or updated in the list when the screen processing mode was *CHANGE should be processed by the loop. For more details, refer to [Screen Modes and Mode Sensitive Commands](#).

*DELETE indicates that only entries that were added or updated in the list when the screen processing mode was *DELETE should be processed by the loop. For more details, refer to [Screen Modes and Mode Sensitive Commands](#).

Values may not be as expected on exit from a selectlist loop due to an extra record being read to determine when to terminate the selectlist loop.

WHERE

For more information, refer to the [Specifying Conditions and Expressions](#) which describes how to specify conditions and expressions.

The WHERE parameter is valid only for working lists. If the WHERE parameter is not specified, all entries in the working list are returned.

After a SELECTLIST/ENDSELECT loop utilizing a where condition, the contents of the fields are unpredictable. The entries matching the where condition should only be processed within the SELECTLIST/ENDSELECT loop.

7.84.2 SELECTLIST Examples

Example 1: Process all "altered" entries from a list named #ORDERLINE. Refer to the DEF_LIST command for more details of lists and list processing.

```
SELECTLIST NAMED(#ORDERLINE) GET_ENTRYYS(*ALTERED)
  << Commands to process the list >>
  << Commands to process the list >>
  << Commands to process the list >>
ENDSELECT
```

Example 2: Process all "selected" entries from a list named #ORDERLINE.

```
SELECTLIST NAMED(#ORDERLINE) GET_ENTRYYS(*SELECT)
  << Commands to process the list >>
  << Commands to process the list >>
  << Commands to process the list >>
ENDSELECT
```

Example 3: Process all entries where quantity is greater than 0 in working list #ORDERLINE.

```
SELECTLIST NAMED(#ORDERLINE) WHERE('#QUANTITY *GT 0')
  << Commands to process the list >>
  << Commands to process the list >>
  << Commands to process the list >>
ENDSELECT
```

7.85 SELECT_SQL

Unlike the standard SELECT command, which uses native IBM i database access, the SELECT_SQL command uses the SQL/400 product to perform database access.

There are two forms of the SELECT_SQL command. The first, which is documented in this section, is heavily structured helping to ensure the SQL is correct and object names that differ between platforms are catered for but it restricts the type of SELECT statements to quite simple ones. The other form of SELECT_SQL is free-format. Any SELECT statement can be entered that the database engine accepts as valid syntax, but LANSA does not attempt to make object names compatible across platforms. These two differences make it more likely that the SQL will not execute as expected across different databases. See [SELECT_SQL Free Format](#) for further information.

The SELECT_SQL command is used in conjunction with the ENDSELECT command to form a "loop" to process one or more rows (records) from one or more tables (files).

For example, the following SELECT_SQL / ENDSELECT loop selects all values of product and quantity from the table ORDLIN and places them, one by one, in a list:

```
----> DEF_LIST NAME(#ALIST) FIELDS(#PRODUCT #QUANTITY)
--> SELECT_SQL FIELDS(#PRODUCT #QUANTITY)
|         USING("SELECT "PRODUCT", "QUANTITY" FROM "MYDTAL
|
|         ADD_ENTRY(#ALIST)
|
---- ENDSELECT
```

The method of implementing SELECT_SQL differs between objects generated as RPG on IBM i and objects generated as C. RPG implements SELECT_SQL in static embedded SQL. C implements SELECT_SQL in a call level interface (CLI) and thus is dynamic. The effect of this distinction is described below where relevant.

Before attempting to use SELECT_SQL you must be aware of the following:

1. To compile functions containing SELECT_SQL commands these licensed products are required:

For IBM i RPG Functions: IBM - SQL DevKit

For C executables: No other products required

If an IBM iRPG application using SELECT_SQL is ported in compiled form from one IBM i to another, it can still be executed, even if the target machine does not have the IBM licensed product installed. However, this situation will cause problems if the need to recompile the application on the target machine ever arises.

2. Information accessed via SELECT_SQL is for read only. If you wish to update information it is often easier to use the standard SELECT command.
3. The SELECT_SQL command is primarily intended for performing complex extract/join/summary extractions from one or more SQL database tables (files) for output to reports, screens or other tables. It is not intended for use in high volume or heavy use interactive applications. With that intention in mind, it must be balanced by the fact that SELECT_SQL is a very powerful and useful command that can vastly simplify and speed up most join/extract/summary applications, no matter whether the results are to be directed to a screen, a printer, or into another file (table).
4. The SELECT_SQL command provides very powerful database extract/join/summarize capabilities that are directly supported by the SQL database facilities. However, the current IBM i implementation of SQL may require and use significant resource in some situations. It is entirely the responsibility of the user to compare the large benefits of this command, with its resource utilization, and to decide whether it is being correctly used. One of the factors to consider is whether the WHERE parameter uses any non-key fields. If it does, then SELECT_SQL will probably be quicker than SELECT. Otherwise SELECT will probably be quicker. This is especially important when developing the program on Visual LANSA first with the intention of also running it on IBM i. This is because Visual LANSA has much fewer performance differences between SELECT and SELECT_SQL.
5. This section assumes that the user is familiar with the SQL 'SELECT' command. This section is about how the SQL 'SELECT' command is accessed directly from RDML functions, not about the syntax, format and uses of the SQL 'SELECT' command.
6. Very limited checking is performed on the correctness of the WHERE, GROUP_BY, HAVING and ORDER_BY parameters.

7. SELECT_SQL does not use the IO Modules/OAMs so it bypasses the repository validation and triggers.
8. When a file is deployed on non-IBM i platforms, by default the table is created using the target partition's data library. But, calls to SELECT_SQL have compiled in the source partition's data library. So if the names are different, you must use the DEFINE_OVERRIDE_FILE Built-In Function to change the table owner.

Error Handling

If an SQL Function is incorrectly quoted by SELECT_SQL, it will cause an error. With SQL Server the error may be "SQL error code 16954...Executing SQL directly; no cursor". Other error codes may occur for the same reason. Other databases will have different error codes.

This occurs when a Function is not known by LANSAs and so the word is presumed to be an identifier and is quoted. The workaround for this is to use the SELECT_SQL Free Format version of the command.

IBM i RPG Functions

If your command is incorrect then there are 2 possible points where it will fail:

- When you compile the RDML function. The SQL command preprocessor will indicate an error in the command and LANSAs will interpret this as a failure to compile the resulting RPG. In this situation the SQL commands embedded in the resulting RPG will have to be examined for error message details.
- At execution time. Even if the application compiles SQL may cause it to fail when the SELECT_SQL command is actually executed. In this case examine all the resulting error messages for the exact cause.

A useful technique when working with SQL is to use interactive SQL to "test case" your command (and its syntax) before compiling it into a SELECT_SQL command.

When dealing with an execution time error, the use of debug on the function will cause SQL to present useful error analysis information. Note that this feature is provided by SQL/400, not by LANSAs, but it will work in conjunction with normal LANSAs debug mode (IBM i only).

C Executables

If your command is incorrect then the following diagnosis is possible:

- When you build the function/component warning messages are displayed. Some of these messages are described in a table below.

- Compiling the function/component will not provide any further information as the SQL is evaluated at execution time. That is, the SQL is dynamic.
- At execution time. Even if the application compiles SQL may cause it to fail when the SELECT_SQL command is actually executed. In this case examine all the resulting error messages for the exact cause.

A useful technique when working with SQL is to use interactive SQL to "test case" your command (and its syntax) before compiling it into a SELECT_SQL command.

When dealing with an execution time error, the use of trace on the function will allow the capture of the exact SQL that the SELECT_SQL command has generated. Open the latest trace file and search for "***ERROR". This will be the same text as in the error messages. Go back 8 lines or so to the "Preparing" message and you will find the SELECT statement that caused the error. You can copy and paste this into interactive SQL to further diagnose the problem.

- When reporting issues with SELECT_SQL to support you must provide the trace file and the generated C source code.

IBM i RPG Functions Only

- Cross-reference information is only taken from the FIELDS and FROM_FILES parameters. References to fields and files embedded in other parameters of this command are not reflected into the LANSAs cross-reference facility in the current release.
- The database's column name must be used when accessing through SQL. C executables can use either the database's column name or the field name that LANSAs knows the column by. This can be different when using Naming Level 0 files. If the field name is used (without the #), LANSAs converts it to a column name at runtime. This allows the name used at execution time to be portable between all platforms. All of the parameters that accept a column name exhibit this behaviour. For example, this RDML using Naming Level 0 file #MYFILE:

```
SELECT_SQL FIELDS(#A$ #B) FROM_FILES((#MYFILE)) WHERE('A_
DISPLAY FIELDS(#A$ #B)
ENDSELECT
```

will work correctly on non-IBM i platforms but will fail on IBM i. Visual LANSAs will issue warning PRC1065 if A_ is not a physical field in one of the files in the FROM_FILES parameter. A portable way to write this so that it executes on all LANSAs platforms is as follows:

```
SELECT_SQL FIELDS(#A$ #B) FROM_FILES((#MYFILE)) WHERE('A$  
  DISPLAY FIELDS(#A$ #B)  
ENDSELECT
```

Visual LANSAs C Functions Only

- The maximum number of SELECT_SQL commands that a single field can appear in is 50
- A LANSAs field used in SELECT_SQL only has one rename in the whole of a Function or Component. Renames are used by VL Other Files and IBM i Other Files. The loading of other files should ensure that a different field is created when a column matches an existing field, and thus the situation should not occur.
- * SELECT/OMIT Criteria in a Logical File specified in the FROM_FILE parameter will be ignored since CLI does not use the Logical File when retrieving data.

The extensive use of the SELECT_SQL command is **not recommended** for the following reasons:

- The SQL access commands are imbedded directly into the RDML function. DBMS access is direct and not done via IOM/OAM access routines. This approach may compromise the use of before and after read triggers and the use of the "thin client" designs implemented via LANSAs/SuperServer.
- If the contents of SELECT_SQL is sourced from a field on a screen then it is possible for an end user to perform more than a select. It is especially easy in the Free Format version where this code is possible:

```
REQUEST FIELD(#ANYSQL)  
Select_Sql Fields(#STD_NUM) Using(#ANYSQL)  
endselect.
```

and the end user could enter this on the screen: "delete from mylib.afe;select count(*) from mylib.afe"

- The use of imbedded SQL features and facilities may introduce platform dependencies into your applications. Not all SQL facilities are supported by

all DBMSs. By bypassing the IOM/OAM associated with the table, you are bypassing the feature isolation it provides. Using SQL features and facilities that are DBMS defined, platform dependent extensions, is solely at the discretion of, and the responsibility of, the application designer.

- Where SELECT_SQL is to be used, you should isolate the use within a specific function, separate from any user interface operations. This will allow the function to be invoked as an "RPC" (Remote Procedure Call) in the client design models.

Messages issued at build time by Visual LANSA

LII0898W Ambiguous. Field #A\$ exists in more than one file and they use different naming algorithms.

This message is reporting about the SQL name that will be used for the field at runtime. There are two further messages which follow this message which provide more detail. The generator decides on the naming algorithm to use based on the following precedence: (1) Older Visual LANSA Files use LANSA mangled names, like #A\$ becomes A_; (2) Naming Level 1 files which use LANSA-defined names, that is, the SQL name is the same as the field name; (3) VL Other File naming or IBM i Other File naming, which ever one appears first in the FROM_FILES parameter.

It is not necessary to change the RDML to eliminate the message. It depends on which file's data you need to access. If the default behaviour is not wanted, then add an SQL source parameter with the real name that is needed.

The following warnings should be eliminated to improve success at runtime and when running on IBM i.

PRC1064 ** WARNING: Name is not a defined field. Correct it for portability.

The field name may be a real column in one of the files and so the select will work, but to work on all LANSA supported databases a field name must be used (without the hash character).

PRC1065 ** WARNING: Field <afield> is not a physical field in any of the files in the FROM_FILES parameter.

LANSA checks if a name specified in SQL is known to LANSA in one of the files in the FROM_FILES parameter. It checks if the name

is a LANSAs name, a converted name or a column rename. It also checks if it is a reserved SQL keyword. If it is none of these, then this warning is displayed:

This can be caused either by using the column name instead of the field name in which case the SQL will still work on Visual LANSAs, or because the field is not correct and so will fail at runtime.

PRC1067 ** Fields A\$ and A_ both resolve to A_ so A_ in SELECT_SQL will be set with Non-IBM i text A_

Two or more fields that resolve to the same name mean that the generated code cannot tell them apart and so a compile error would occur. So, for backward compatibility, SELECT_SQL uses a fixed literal value so the compile will succeed. But, this may not execute on IBM i. Change your code so that it does not use both these matching fields in the one function.

For example, the column name has been fixed at A_, so it will not run on IBM i. Use A\$ instead.

Portability Considerations When using multiple platforms, you must take into consideration the length of the field names used by each of the platforms. Refer to the [WHERE](#) parameter.

Do NOT use this command to connect from Visual LANSAs to a database on the IBM i. If you use the SELECT_SQL command to connect from Visual LANSAs to an IBM i Database, it will access the Database on the PC and not on the IBM i. For this type of connection, you should use a remote procedure call (i.e call_server_function).

Also See

[7.85.1 SELECT_SQL Parameters](#)

[7.85.2 SELECT_SQL Column Names versus Column Values](#)

[7.85.3 SELECT_SQL Examples](#)

[7.85.4 SELECT_SQL References](#)

[7.85.5 SELECT_SQL Coercions](#)

Required

*SELECT_SQL --- FIELDS ----- field name --- *SAME ----*

----->

| *SQL field source* |
----- 1000 max -----

>-- *FROM_FILES* ----- *file name* -- *correlation* ---

>

| |
----- 20 max-----

Optional

>-- *WHERE* ----- '*SQL where condition*' -----

>

>-- *GROUP_BY* ----- '*SQL group by clause*' -----

-->

>-- *HAVING* ----- '*SQL having condition*' -----

>

>-- *ORDER_BY* ----- '*SQL order by parameter*' -----

-->

>-- *DISTINCT* ----- **NO* ----->
 **YES*

>-- *IO_STATUS* ----- *field name* ----->
 **STATUS*

>-- *IO_ERROR* ----- **ABORT* -----|
 **NEXT*
 **RETURN*
 label

7.85.1 SELECT_SQL Parameters

DISTINCT

FIELDS

FROM_FILES

GROUP_BY

HAVING

IO_ERROR

IO_STATUS

ORDER_BY

WHERE

See also [7.85.2 SELECT_SQL Column Names versus Column Values](#)

FIELDS

Specifies the columns (fields) and their associated "SQL source" or function.

Fields of type BLOB and CLOB are not supported in the SELECT_SQL command. If one is specified a fatal error will occur when the command is compiled.

All columns nominated by this parameter must be defined in the current function or in the LANSAs data dictionary as valid RDML variables.

For each column specified an optional field "SQL source" may be nominated. This field has a maximum length of 50 characters.

When the source is not specified, the default value of *SAME (same as column name) is used. This means that the column name in the function and its "source" in the SQL table are assumed to be the same. When this value is used the column must be defined as a valid real column in one (or more) of the tables nominated in the FROM_FILES parameter.

For example:

```
SELECT_SQL FIELDS((#CUSTNAM)) FROM_FILES(CUSTOMST)
```

indicates that the column named CUSTNAM is to be extracted from the table CUSTOMST and its value returned into the RDML function into the field called #CUSTNAM. This example uses the *SAME default. But the example:

```
SELECT_SQL FIELDS((#CUSTNO CUSTNAM)) FROM_FILES(CUSTOMST)
```

indicates that column named CUSTNAM is to be extracted from the table CUSTMST and its value returned into the RDML function into the field called #CUSTNO.

And, the further example:

```
SELECT_SQL FIELDS((#SHORTNAME 'SUBSTR(CUSTNAM,3,10)'))
```

indicates that a substring of column CUSTNAM (from the SQL table) is to be returned into the RDML function field #SHORTNAME.

And, another example where two files are being joined and the column CUSTNAM is in both tables a correlation is used to clarify which table, CUSTMST or CUSTMST2 to obtain the data from:

```
SELECT_SQL FIELDS((#CUSTNAM 'A.CUSTNAM')) FROM_FILES((CUSTMST A) (CUSTMST2 B)) WHERE('A.CUSTID = B.CUSTID')
```

And, the final example:

```
SELECT_SQL FIELDS(#DEPARTMENT (#VALUE1 'AVG(SALARY)') (#VALUE2 'SUM(SALARY)') (#VALUE3 'MAX(SALARY)'))
```

indicates that SQL table column DEPARTMENT is to be returned into RDML variable #DEPARTMENT, the average of SQL table field SALARY is to be returned into RDML variable #VALUE1, the total into #VALUE2, and the maximum into #VALUE3.

FROM_FILES

Refer to [Specifying File Names](#).

Note: Up to 20 file (table) names can be specified for use by this command.

Note: When accessing Other Files that are in other databases LANSAs locates the database connection information that was used to load the Other File into LANSAs. This can be further refined by using the DEFINE_DB_SERVER and CONNECT_FILE BIFs

For each file name specified an optional field "correlation" may be nominated.

When the correlation is not specified, the default of *SAME (IE same as file name) is assumed. This means that when referring to a column in a specific table the actual table name must be used. If a correlation name is used the correlation name must be used to identify a column from a specific table.

Examples

```
... FROM_FILES((ORDLIN) (ORDDTL)) WHERE('ORDLIN.CUSTNO = C
```

If correlations were used this statement could be written as:

```
... FROM_FILES((ORDLIN A) (ORDDTL B)) WHERE('A.CUSTNO = B.C
```

Portability Considerations Visual LANSAs does not use @#\$ in table names. This conversion is done for the FROM_FILES parameter, but not for table names in the other parameters, e.g. the WHERE parameter. So, in order that the SQL can work on all LANSAs platforms, correlations should always be used as in the second example above.

Visual LANSAs provides access to multiple databases using Visual LANSAs Other Files. Visual LANSAs Other files can be used in SELECT_SQL, but they must all be from the same database. If a Visual LANSAs Other File is in the same database as a LANSAs file, then the two can be used in the same SELECT_SQL command

WHERE

You must enclose the SQL_SELECT WHERE clause in quotes as shown here:

```
... WHERE('EMPNO < "A9999"')
```

```
... WHERE('NOT EMPNO LIKE "%a"')
```

The where clause may contain either LANSAs field names, or column names. (Refer to the FIELDS Parameter for more details.)

The SQL language uses double quotes to surround identifiers that might otherwise be interpreted as SQL syntax. LANSAs leaves the contents of double-quoted text untouched. Note, this is the double-quote character ("), not two single quotes (').

Portability Considerations LANSAs field names in the WHERE parameter will be generated as double-quoted column names into the SQL statement, as long as the field is recognized as being from one of the files in the

FROM_FILES parameter. Note that the column name is not always the same as the field name. This is often the case for files on Other Files, but also happens for certain field names on LANSAs files. Refer to [Convert Special Characters in Field Names](#)

An exception to the previous paragraph is when an unquoted LANSAs field name conflicts with SQL keywords. In this case the field name is NOT converted. Refer to *SQL/ODBC Grammar: Keyword Conflicts* for more details.

For example, SECTION is a LANSAs field in SECTAB. This is created as the column S_SECTION.

If the WHERE parameter was written as

WHERE('SECTION = "1"') then SECTION would not be converted and thus will cause an SQL syntax error at runtime.

A workaround for this is to use a correlation so that LANSAs knows your intention is to access the column and not use it as a SQL keyword. An example would be:

```
FROM_FILES((SECTAB SEC)) WHERE('SEC.SECTION = "
```

If selecting from two or more files that have the same LANSAs field, the column name may differ between the files. Refer to [Convert Special Characters in Field Names](#). In this case, you, the developer, have two ways to control the SQL WHERE clause that LANSAs generates. The first is to use a correlation (refer to the FROM_FILES parameter) so that LANSAs renames it according to the rules of that file. The second is to use the column name and enclose it in double quotes.

If using field names in the WHERE parameter, it is recommended that you leave space around the field name so that LANSAs can recognise the field names and convert them appropriately. For example, WHERE('a=b') should be instead be WHERE ('a = b')

Visual LANSAs allows a single field name to be specified instead as shown here:

```
CHANGE FIELDS(#SELECTION) TO('STATE = "NSW"')
SELECT_SQL FIELDS(#STATE) FROM_FILES((#STATES)) WHERE(#SELECTION)
  DISPLAY FIELDS(#STATE)
ENDSELECT
```

The contents of the field are used as the WHERE clause and the following

needs to be considered:

- A build warning will be generated if a single field name is used in LANSAs for i RPG Functions. An error will occur at execution time. Code using this facility can be conditioned so that it is not executed in this environment. See the *OPNQRYF command for an alternate programming method and how to write portable code
- The field name STATE is not preceded by a # (hash) symbol in this example. This is because the actual selection request is being made to the SQL database, not to LANSAs. That is, the actual column name must be used. Visual LANSAs renames column names that contain @, # or \$ and replaces them with an underscore in Naming level 0 files, but this does not occur on LANSAs for i, thus the code is portable provided that the LANSAs field name is used, not the actual column name. E.g. if the Field is CUST\$NAM then this should be used, not CUST_NAM. Visual LANSAs will change CUST\$NAM to CUST_NAM at runtime (Naming Level 0 file). LANSAs for i will leave it as it is.
- Variable comparison values as in :KARTIC will not be replaced. Instead the value must be concatenated into the #SELECTION field.

WHERE clause hints

When searching for data using the like condition, characters with special meaning to SQL need to be escaped if they need to be taken literally. For example, the character '_' matches any character. To literally match '_' then the following syntax needs to be used. This will find all states that start with 'B_':

```
CHANGE FIELDS(#SELECTION) TO('STATE LIKE "B!_%" ESCAPE "!")
```

Note: This nominates the exclamation mark as the escape character. Any "normal" character not greater than 127 in the ASCII table can be used. (Characters %,_,[do not work on all DBMS systems and so are not recommended.)

This has been tested on ASA, DB2400, SQL Server, and Oracle. The only exception is MS Access, where instead you need to use [] around the character to be escaped. For example: WHERE STATE LIKE 'B[_]%'

Using a Field for Variable Comparison Values

Place a colon (:) immediately in front of the field name, without any spaces separating the colon from the field name to indicate that the name in the WHERE clause should be used. (If the field name is more than six characters long, you will get unpredictable results in LANSAs for i RPG Functions.)

Portability Considerations Fields used to contain variable comparison values in the WHERE clause are not translated by LANSAs. Therefore, when using fields in this way, your field names must be six (6) characters or less. (This is because fields are used with their actual name in the generated RPG on LANSAs for i.) LANSAs for i RPG Functions allow a space between the colon and the field name, but this does not work in generated C code. For portability do not leave a space between the colon and the field name.

If your code will run in a LANSAs for i RPG function, and you have field names that are longer than six characters that are to be used in the WHERE clause, you will need to define a work field for these names with a name that is six characters or less, as in the example below.

Example:

```
DEFINE    #KARTIC REFFLD (#ARTICO)
DEFINE    #KADTRG REFFLD (#MADTRG)

... WHERE ('ARTICO = :KARTIC AND MADTRG > :KADTRG')
```

For further details about specifying conditions, refer to [Specifying Conditions and Expressions](#). For further information about the structure of this clause, refer to the SQL guides.

RDMLX IBM i Other Files with Unicode Fields

SQL on IBM i cannot compare a graphic unicode field directly to a string literal or a character column; a conversion error occurs.

There are two ways of converting the expression to Unicode to avoid the conversion error:

1. Use a LANSAs field for comparison. For example, WHERE('MYUNIGRPH = :STD_TEXT').
2. Pass the literal as a Unicode (UX'ssss') literal. For example, instead of:

```
WHERE('UNIFLD LIKE 'C%')
```

```
try
```

```
WHERE('UNIFLD LIKE UX"00430025").
```

For further details, refer to the IBM manual *DB2 UDB for IBM SQL Reference*.

GROUP_BY

Is used to find the characteristics of groups of rows rather than individual rows. Grouping does not mean sorting. Grouping puts each selected row in a group which SQL processes to derive characteristics of the group.

Specify the column(s) you want to group the selected rows by. If more than one column is specified, commas must be used to separate the data. For example, `GROUP_BY('EMPTSYEAR, EMPTSWEEK')`.

HAVING

Is used to specify a search condition for the groups selected based on a `GROUP_BY` clause. The `HAVING` parameter says that you want only those groups that satisfy the condition in the clause. That is, the `HAVING` clause tests the properties of each group not the properties of the individual rows in the group.

The `HAVING` clause can contain the same kind of search condition that can be used in the `WHERE` parameter.

ORDER_BY

Use this parameter to specify the order you want the selected rows retrieved. The order by parameter can be used the same way as the `GROUP_BY` parameter.

Specify the name of the column or columns SQL should use when retrieving the rows in a column. If more than one column is specified, commas must be used to separate the data. For example, `ORDER_BY('SURNAME, GIVENAME')`.

DISTINCT

Specify `*YES` to this parameter if duplicate rows are not required in the result of the `SELECT_SQL`.

Specify `*NO` if duplicate rows are required in the result table.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the I/O return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

Refer to [I/O Command Return Codes Table](#) for values.

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

7.85.2 SELECT_SQL Column Names versus Column Values

The basic rule is:

- if a name is preceded by a '#' it means the Column Name should be used
- if its preceded by a ':' it means the Column Value should be used.

The only exception to this is in the WHERE and HAVING parameters when the #Field is the ONLY value in the parameter. In that case, it means use the Column Value.

The secondary rule is that if an identifier does not have a '#' or ':' then it will be interpreted as a column name unless its also an SQL Name, in which case it will be left exactly as typed in the generated code. The Visual LANSA Editor will display the following warning if this is the case:

Ambiguous. Token <name> is an SQL keyword and a LANSA field. If it's a LANSA field, prepend a '#' to the field.

Following are some examples to help explain it more fully.

When using a column name that is an SQL keyword, LANSA will not convert it. So it must be specified explicitly as either the mangled name, LANSA Name or Long Name, depending on how the table has been created. E.g. #SECTION. This is mangled to S_CTION and its long name may be set to SectionCode.

If the file is using mangled names or long names then this code will not work

```
Select_Sql Fields(#SECTION) From_Files((PSLMSTX2))
Group_By(SECTION)
Add_Entry
Endselect
```

The SQL would be:

```
SELECT "SectionCode" FROM "EVDEXLIB"."PersonnelMaster2" GROUP
BY "SECTION"
```

The Group_By(SECTION) will be left as it is, which will not match the actual column name - S_CTION or SectionCode.

To fix this code in the most flexible manner (See **Note 2** following) prepend a '#' to the name as in:

```
Select_Sql Fields(#SECTION) From_Files((PSLMSTX2))
```

```
Group_By(#SECTION)
Add_Entry
Endselect
```

The SQL for this would be:

```
SELECT "SectionCode" FROM "EVDEXLIB"."PersonnelMaster2" GROUP
BY "SectionCode"
```

But if the field name is not an SQL keyword like EMPNO here, it WILL automatically convert the Field name to the actual column name, with or without the '#':

```
Select_Sql Fields(#EMPNO) From_Files((PSLMSTX2)) Group_By(EMPNO)
Add_Entry
Endselect
```

If the Long Name for EMPNO is EmployeeNumber, and PSLMSTX2 allows long names then the Group_By EMPNO will be resolved to EmployeeNumber. The SQL would be:

```
SELECT "EmployeeNumber" FROM "EVDEXLIB"."PersonnelMaster2"
GROUP BY "EmployeeNumber"
```

Note 1: '#Field' means use the column name in the generated SQL in all parameters except when it's the ONLY value in WHERE or HAVING, in which case it will generate the runtime contents of the Field. The Free Format version of SELECT_SQL - the USING parameter - is not included in this. It has its own semantics described in [SELECT_SQL Free Format Parameters](#).

Note 2: 'Flexible manner' in the sense that if the old style Windows mangling of column names is being used then the generator will automatically use "S_SECTION" for Windows & Linux and "SECTION" on IBM i.

SELECT_SQL Examples

[Using SELECT_SQL With the DISTINCT Option](#)

[Using SELECT_SQL With Calculations](#)

[Using SELECT_SQL With AND and OR Operators](#)

[Using SELECT_SQL With the BETWEEN Operator](#)

Using SELECT_SQL With the DISTINCT Option

This example demonstrates how to use the SELECT_SQL command with the DISTINCT option to eliminate duplicate field values. The use of the standard SELECT_SQL command without any extra options is also demonstrated.

```
DEF_LIST  NAME(#EMPBROWSE) FIELDS(#NDSTEMPNO #DSTEMPN
DEFINE    FIELD(#HEADING1) TYPE(*CHAR) LENGTH(79) INPUT_ATI
DEFINE    FIELD(#NDSTEMPNO) REFFLD(#EMPNO) COLHDG('Employ
DEFINE    FIELD(#DSTEMPNO) REFFLD(#EMPNO) COLHDG('Employee
DEFINE    FIELD(#ENTRYNO) TYPE(*DEC) LENGTH(5) DECIMALS(0)
```

```
CHANGE    FIELD(#HEADING1) TO('"This function uses SELECT_SQL fr
```

```
BEGIN_LOOP
EXECUTE    SUBROUTINE(NOTDISTINC)
EXECUTE    SUBROUTINE(DISTINCT)
DISPLAY    FIELDS(#HEADING1) DESIGN(*DOWN) IDENTIFY(*NOID)
END_LOOP
```

```
SUBROUTINE NAME(NOTDISTINC)
CLR_LIST  NAMED(#EMPBROWSE)
CHANGE    FIELD(#DSTEMPNO) TO(*NULL)
SELECT_SQL FIELDS(#EMPNO) FROM_FILES((PSLSKL))
CHANGE    FIELD(#NDSTEMPNO) TO(#EMPNO)
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
ENDROUTINE
```

```
SUBROUTINE NAME(DISTINCT)
CHANGE    FIELD(#ENTRYNO) TO(1)
SELECT_SQL FIELDS(#EMPNO) FROM_FILES((PSLSKL)) DISTINCT(*Y
GET_ENTRY NUMBER(#ENTRYNO) FROM_LIST(#EMPBROWSE)
```

```

CHANGE FIELD(#DSTEMPNO) TO(#EMPNO)
UPD_ENTRY IN_LIST(#EMPBROWSE)
CHANGE FIELD(#ENTRYNO) TO('#ENTRYNO + 1')
ENDSELECT
ENDROUTINE

```

Using SELECT_SQL With Calculations

This example demonstrates how calculations can be used on data retrieved by the SELECT_SQL command.

```

DEF_LIST NAME(#EMPBROWSE) FIELDS(#SURNAME #SALARY #STD_AMNT)
DEFINE FIELD(#HEADING1) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(*NONE)
DEFINE FIELD(#HEADING2) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(*NONE)
DEFINE FIELD(#HEADING3) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(*NONE)

OVERRIDE FIELD(#STD_AMNT) COLHDG('Salary + 10%')

CHANGE FIELD(#HEADING1) TO('"This function uses SELECT_SQL fr
CHANGE FIELD(#HEADING2) TO('"This shows a list of employee surnam
CHANGE FIELD(#HEADING3) TO('"This can be done with one SELECT_

BEGIN_LOOP
CLR_LIST NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#SURNAME #SALARY (#STD_AMNT 'SALARY *
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
DISPLAY FIELDS(#HEADING1 #HEADING2 #HEADING3) DESIGN(*I
END_LOOP

```

Using SELECT_SQL With AND and OR Operators

This example demonstrates how the SLECT_SQL command can be used with AND and OR operators to conduct more complex queries.

```

DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #ADDRESS2 #SALARY)
DEFINE FIELD(#HEADING1) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(*NONE)
DEFINE FIELD(#HEADING2) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(*NONE)
DEFINE FIELD(#HEADING3) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(*NONE)

CHANGE FIELD(#HEADING1) TO('"This function uses SELECT_SQL fr

```

```
CHANGE FIELD(#HEADING2) TO("This lists all employees who either h
CHANGE FIELD(#HEADING3) TO("or who live in SEVEN HILLS. This
```

```
BEGIN_LOOP
CLR_LIST NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#EMPNO #SURNAME #ADDRESS2 #SALARY) FI
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
DISPLAY FIELDS(#HEADING1 #HEADING2 #HEADING3) DESIGN(*I
END_LOOP
```

Using SELECT_SQL With the BETWEEN Operator

This example demonstrates the use of the SELECT_SQL command with the BETWEEN operator. The BETWEEN operator can be used in the WHERE clause to retrieve data between specified values. It can also be used to retrieve data excluding that between specified values.

```
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #SALARY)
DEFINE FIELD(#HEADING1) TYPE(*CHAR) LENGTH(079) INPUT_A
DEFINE FIELD(#HEADING2) TYPE(*CHAR) LENGTH(079) INPUT_A
DEFINE FIELD(#HEADING3) TYPE(*CHAR) LENGTH(079) INPUT_A
DEF_COND NAME(*AS400) COND('*CPUTYPE = AS400')
```

```
CHANGE FIELD(#HEADING1) TO("EXAMPLE 1: Select all employees ›
CHANGE FIELD(#HEADING2) TO(*BLANKS)
CHANGE FIELD(#HEADING3) TO("This can be done with one SELECT_
```

```
BEGIN_LOOP
CHANGE FIELD(#HEADING1) TO("EXAMPLE 1: Select all employees ›
CLR_LIST NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#EMPNO #SALARY) FROM_FILES((PSLMST)) W
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
```

```
EXECUTE SUBROUTINE(DISP)
CHANGE FIELD(#HEADING1) TO("EXAMPLE 2: Select all employees ›
CLR_LIST NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#EMPNO #SALARY) FROM_FILES((PSLMST)) W
ADD_ENTRY TO_LIST(#EMPBROWSE)
```

```
ENDSELECT  
EXECUTE SUBROUTINE(DISP)  
END_LOOP
```

```
SUBROUTINE NAME(DISP)  
DISPLAY FIELDS(#HEADING1 #HEADING2 #HEADING3) DESIGN(*I  
ENDROUTINE
```

For more examples of the SELECT_SQL command please see 'All About SELECT_SQL' in The Set Collection.

7.85.4 SELECT_SQL References

SAA Structured Query Language/400 Reference (SC41-9608)
SAA Structured Query Language/400 Programmers Guide (SC41-9609)

Coercion: Best attempt to take a value of one type and make some sense of it in another type. For example, packed 1234 becomes Alpha "1234" when viewed on a Form. In this case the underlying packed data has been converted to a string and then assigned to the Alpha field. This is in contrast to a LANSAL OVERLAY, where no conversion is performed and the alpha would contain the same binary data as the Packed field, that is, the Alpha would NOT display the number when viewed on a Form. Any reliance on coercions must be thoroughly tested for the entire range of expected source values on all databases that the application will run on.

RDML Field types in an RDML Function are all interchangeably coercible. The only platform difference is that an overflow or underflow on Visual LANSAL platforms sets the field value to 0. On IBM i, an ABEND occurs. For example, Assigning 123 to a Packed(7,7).

7.86 SET_ERROR

The SET_ERROR command is used to set an error against a field within a BEGINCHECK / ENDCHECK validation block.

Normally the SET_ERROR command is used when a set of RDML commands other than the standard commands RANGECHECK, DATECHECK, VALUECHECK, etc is used to validate a field.

Also See

[7.86.1 SET_ERROR Parameters](#)

[7.86.2 SET_ERROR Examples](#)

Required

```
SET_ERROR ---- FOR_FIELD ---- field name -----  
----->
```

```
      |expandable group expression |  
      ----- 100 max -----
```

Optional

```
>-- MSGTXT ----- *NONE ----->  
      message text
```

```
>-- MSGID ----- *NONE ----->  
      message identifier
```

```
>-- MSGF ----- *NONE . *LIBL ----->  
      message file . library name
```

```
>-- MSGDTA ----- substitution variables -----|  
      |expandable group expression |  
      ----- 20 max -----
```

7.86.1 SET_ERROR Parameters

FOR_FIELD

MSGDTA

MSGF

MSGID

MSGTXT

FOR_FIELD

Specifies the name of the field(s) which are to have an error set against them. The next screen presented to the user will have the field(s) displayed in reverse image with the cursor positioned to the first field in error. An expandable group expression can be entered in this parameter.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this:

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAP aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA("BOLTS  " #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.86.2 SET_ERROR Examples

Example 1: The following 2 validation blocks are functionally identical.

The first uses a "standard" validation check:

```
BEGINCHECK  
  
CONDCHECK FIELD(#QUANTITY) COND('#QUANTITY >= 0) AND (#C  
  
ENDCHECK
```

The second uses the SET_ERROR command to achieve exactly the same result:

```
BEGINCHECK  
  
IF      COND('#QUANTITY < 0) OR (#QUANTITY > (#MEASURE * 1.4  
SET_ERROR FOR_FIELD(#QUANTITY) MSGTXT('Quantity exceeds top n  
ENDIF  
  
ENDCHECK
```

Example 2: The following 2 validation blocks are also functionally identical.

```
BEGINCHECK  
  
FILECHECK FIELD(#PRODNO) USING_FILE(PRODUCT) MSGTXT('Pro  
  
RANGECHECK FIELD(#ORDNUM) RANGE(A000000 Z999999) MSGTXT  
  
RANGECHECK FIELD(#QUANTITY) RANGE(1 9999) MSGTXT('Quantity  
  
ENDCHECK
```

and

```
BEGINCHECK  
  
CHECK_FOR IN_FILE(PRODUCT) WITH_KEY(#PRODNO)  
IF_STATUS IS_NOT(*EQUALKEY)  
SET_ERROR FOR_FIELD(#PRODNO) MSGTXT('Product number not foun  
ENDIF
```

```
IF      '#ORDNUM < A000000) OR (#ORDNUM > Z999999)'
SET_ERROR FOR_FIELD(#ORDNUM) MSGTXT('Order number is not in r
ENDIF
```

```
IF      '#QUANTITY < 1) OR (#QUANTITY > 9999)'
SET_ERROR FOR_FIELD(#QUANTITY) MSGTXT('Quantity ordered must
ENDIF
```

```
ENDCHECK
```

7.87 SET_MODE

The SET_MODE command is used to set the screen mode for a subsequent "mode sensitive" command. Refer to [Screen Modes and Mode Sensitive Commands](#) for details of "mode sensitive" commands and screen mode processing techniques. Mode sensitive commands include DISPLAY, ADD_ENTRY, INZ_LIST and UPD_ENTRY.

Also See

[7.87.1 SET_MODE Parameters](#)

[7.87.2 SET_MODE Examples](#)

Required

```
SET_MODE ----- TO ----- *DISPLAY -----  
---|  
    *ADD  
    *CHANGE  
    *DELETE
```

7.87.1 SET_MODE Parameters

TO

TO

Specifies the mode to which the system is to be set. Allowable values are *DISPLAY, *ADD, *CHANGE and *DELETE.

7.87.2 SET_MODE Examples

Example 1: Set the screen to display mode before displaying a set of fields to the user:

```
SET_MODE TO(*DISPLAY)
DISPLAY FIELDS(#ORDER #CUSTNO #ADDRL1 #ADDRL2 #POSTCD
```

Example 2: Set the mode to add before initializing a list with 20 entries:

```
SET_MODE TO(*ADD)
CHANGE FIELD(#ORDERLINE) TO(*NULL)
INZ_LIST NAMED(#ORDERLINE) NUM_ENTRYYS(20)
```

7.88 SKIP

The SKIP command is used to skip to a nominated line on a report prior to printing information on the report.

Also See

[7.88.1 SKIP Parameters](#)

[7.88.2 SKIP Examples](#)

Optional

SKIP ----- TO_LINE ----- 1 ----->
decimal value

>-- ON_REPORT ---- 1 -----|
report number 1 -> 8

7.88.1 SKIP Parameters

[ON_REPORT](#)

[TO_LINE](#)

TO_LINE

Specifies the line on the report which is to be skipped to. The default value is 1. Otherwise specify a line number between 1 and the overflow line associated with the report. See the DEF_REPORT command for details of the overflow line.

ON_REPORT

Specifies the report which is to be used by this command. Up to 8 reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this parameter is report number 1.

7.88.2 SKIP Examples

Skip to line 32 on report 1 prior to printing a line called #TOTAL.

```
SKIP    TO_LINE(32)  
PRINT  LINE(#TOTAL)
```

7.89 SORT_LIST

The SORT_LIST command is used to sort a list into a nominated sequence.

The list specified must be a working list (used to store information within a program). It is not possible to use the SORT_LIST command against a browse list (used for displaying information at a workstation).

Sorting a Static Working List or a Dynamic Working List with all its entries in a single block of memory is accomplished by sorting the entries in place.

When a Dynamic Working List has its entries across multiple blocks of memory, Visual LANSA performs the sort by allocating a single block of memory; filling the block with pointers that address each entry in the working list and then sorting the block of pointers.

While this technique enables the SORT_LIST command to work with Dynamic Working Lists managing multiple blocks of memory, the SORT_LIST command has a limitation – it must allocate a single block of memory that can contain a pointer to each entry in the working list:

- For the IBM i where each pointer is 16 bytes and the maximum block size is 16MB, only 1MB of entries can be sorted.
- For 32-bit Windows system where a pointer is 4 bytes and it is theoretically possible to allocate a 1GB block of memory, the limit is much higher.

Refer to the [7.23 DEF_LIST](#) command for further information of lists and list processing.

Also See

[7.89.1 SORT_LIST Parameters](#)

[7.89.2 SORT_LIST Examples](#)

Optional

```
SORT_LIST ---- NAMED ----- *FIRST -----  
---->
```

name of list

```
BY_FIELDS ---- name of field --- *ASCEND -----
```

|

```
|           *DESCEND |  
| expandable group expression |  
----- 100 maximum -----
```


7.89.1 SORT_LIST Parameters

BY_FIELDS

NAMED

NAMED

Specifies the name of the list which is to be sorted.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used. This list must have the TYPE(*WORKING) parameter to be valid.

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command and must have the TYPE(*WORKING) parameter.

BY_FIELDS

Specifies the fields whose contents are to be used to order the list and optionally whether the ordering is to be in ascending or descending sequence. An expandable group expression may be entered in this parameter.

Fields nominated in this parameter must be defined in the list nominated in the NAMED parameter.

Where no order sequence is specified for a field, ascending order (*ASCEND) is assumed.

7.89.2 SORT_LIST Examples

Example 1: Sort the entries in a list called #TOTALS by field #COMP into ascending order:

```
DEF_LIST NAME(#TOTALS) FIELDS(#COMP #DEPT #DIV #SALES) TY
SORT_LIST NAMED(#TOTALS) BY_FIELDS(#COMP)
```

Example 2: Sort the entries in a list called #TOTALS by fields #COMP and #DEPT into ascending order:

```
DEF_LIST NAME(#TOTALS) FIELDS(#COMP #DEPT #DIV #SALES) TY
SORT_LIST NAMED(#TOTALS) BY_FIELDS(#COMP #DEPT)
```

Example 3: Sort the entries in a list called #TOTALS by field #SALES (descending order), then fields #COMP #DEPT and #DIV in ascending order:

```
DEF_LIST NAME(#TOTALS) FIELDS(#COMP #DEPT #DIV #SALES) TY
SORT_LIST NAMED(#TOTALS) BY_FIELDS((#SALES *DESCEND) #CC
```

7.90 SPACE

The SPACE command is used to space a nominated number of lines prior to printing information on the report.

Also See

[7.90.1 SPACE Parameters](#)

[7.90.2 SPACE Examples](#)

Optional

SPACE ----- NUM_LINES ---- 1 -----

->

decimal value

>-- ON_REPORT ---- 1 -----|

report number 1 -> 8

7.90.1 SPACE Parameters

NUM_LINES

ON_REPORT

NUM_LINES

Specifies the number of lines that are to be spaced on the report. The default value is 1. Otherwise specify a value between 1 and 100.

ON_REPORT

Specifies the report which is to be used by this command. Up to 8 reports can be produced by a function at one time. Each report is identified by a number in the range 1 to 8. The default value for this parameter is report number 1.

7.90.2 SPACE Examples

Example 1: Space 1 line before printing a line called #TOTAL:

```
SPACE  
PRINT  LINE(#TOTAL)
```

Example 2: Space 5 lines before and after printing a line called #TOTAL:

```
SPACE  NUM_LINES(5)  
PRINT  LINE(#TOTAL)  
SPACE  NUM_LINES(5)
```

7.91 SUBMIT

The SUBMIT command is used to submit a call to a program to batch, to a process/function to batch or to start a Form if executing on Windows.

Optionally parameters may be passed to the program or the LANSAs process/function.

If the submitted call is to a process/function or a Form it is also possible to exchange information with it using the exchange list. Refer to the EXCHANGE command for more details of the exchange list and how it is used.

Portability Refer to parameters: [JOBBD](#) , [JOBQ](#) , [PARM](#) , [PGM](#) and **Considerations** [OUTQ](#) and [Specifying File Names in I/O Commands](#).

Also See

[7.91.1 SUBMIT Parameters](#)

[7.91.2 SUBMIT Comments / Warnings](#)

[7.91.3 SUBMIT Examples](#)

Optional

```
SUBMIT ----- PGM ----- *NONE -----  
>
```

```
    pgm name . *LIBL  
    pgm name . library name
```

```
>-- PROCESS ----- *NONE ----->  
    process or Form name
```

```
>-- FUNCTION ----- *FIRST ----->  
    function name or *FORM if executing a
```

Form

```
>-- PARM ----- list of parameters ----->  
    | expandable group expression |  
    ----- 20 maximum -----
```

```
>-- EXCHANGE ----- field name ----->  
    | expandable group expression |
```

| |
----- 100 max -----

>-- JOB ----- *PGMPRO ----->
job name

>--- JOB D ----- Q B A T C H . * L I B L ----->
*USRPRF
job description . *LIBL
job description . library name

>--- JOB Q ----- *JOB D ----->
job queue . *LIBL
job queue . library name

>--- O U T Q ----- *JOB D -----|
*CURRENT
*USRPRF
*DEV
output queue . *LIBL
output queue . library name

7.91.1 SUBMIT Parameters

EXCHANGE

FUNCTION

JOB

JOBD

JOBQ

OUTQ

PARM

PGM

PROCESS

PGM

Specifies the name of the program which is to be invoked in batch. This parameter is a qualified name. Either a program name or a process name (but not both) must be specified on this command. If required the library in which the program resides can also be specified. If no library name is specified, library *LIBL is assumed which indicates the execution time library list of the **batch job** should be searched to find the program.

Portability

Considerations The submit of 3GL programs is only supported on IBM i for compatibility with existing RDML Code. As such, only RDML fields are supported in the PARM parameter that may be used for the submit of 3GL programs.

Not supported in the current version of Visual LANSA but will be supported in a future release. A build warning will be generated if used and an error will occur at execution time. Code using this facility can be made conditional so that it is not executed in this environment.

PROCESS

Specifies the name of the LANSA process or Form that is to be invoked. Either the PGM parameter or the PROCESS parameter (but not both) must be specified.

Portability Considerations

A Form can only be executed on Windows.

FUNCTION

Optionally specifies the function within the nominated process that should be invoked in batch. If this parameter is not specified a default value of *FIRST is assumed that indicates that the first function (alphabetically) associated with the nominated process should be invoked.

When a Form is specified for the PROCESS parameter, this value must be *FORM.

PARM

Is optional, and if specified defines a list of parameters which are to be passed to the batch program or process. The parameters must correspond in number and type to those expected by the program or process. This is **not checked** by LANSAs.

Parameters specified may be alphanumeric or numeric literals, field names, an expandable group expression, system variables or process parameters. They are passed to the called program or process with the same type and length attributes as they are defined within LANSAs.

When passing numeric parameters to a process they must always be in packed decimal format. This rule does not necessarily apply to user application programs written in other languages such as RPG or COBOL.

The passing of parameters to a process is not recommended. Use the exchange list instead as a more flexible means of passing information to the batch process. Refer to the EXCHANGE parameter of this command and to the EXCHANGE command in this guide for more details of the exchange list and exchange list processing.

Portability Considerations Not supported in the current release of Visual LANSAs and not expected to be in future releases.

EXCHANGE

Optionally specifies the name of the field(s) whose value(s) are to be exchanged, or the name of a group that defines the field(s) whose value(s) are to be exchanged, with the batch function.

For details of how field and group names can be specified in this parameter,

refer to [Field Groups and Expandable Groups](#). For more details of how information is exchanged between functions, refer to the [EXCHANGE](#) command.

Use of the EXCHANGE parameter is only valid when submitting a process/function/form to batch. The parameter is ignored when submitting a program to batch.

The EXCHANGE parameter on this command is provided for convenience only. Using it is identical to using one or more EXCHANGE commands before the SUBMIT command. Thus:

```
SUBMIT PROCESS(PROC01) FUNCTION(FUN1) EXCHANGE(#A #B #C)
is functionally identical to:
```

```
EXCHANGE FIELDS(#A #B #C #D)
SUBMIT PROCESS(PROC01) FUNCTION(FUN1)
```

which is functionally identical to:

```
EXCHANGE FIELDS(#A)
EXCHANGE FIELDS(#B)
EXCHANGE FIELDS(#C)
EXCHANGE FIELDS(#D)
SUBMIT PROCESS(PROC01) FUNCTION(FUN1)
```

Note that the exchange list is **cleared of all entries** after the SUBMIT command has completed execution.

JOB

Optionally specifies the name that is to be assigned to the batch job when it is submitted. The name specified must conform to IBM i naming conventions. Refer to the *LANSA Application Design Guide* for LANSA's naming conventions.

If this parameter is not specified, default value *PGMPRO is assumed. This indicates that the job submitted should have the same name as either the program specified in the PGM parameter or the process/form specified in the PROCESS parameter (which ever is used).

JOBID

Specifies the name (and optionally library) of the job description that is to be used to submit the job to batch.

To submit a job on the IBM i a job description is always required. The details of what a job description is and how it is used are beyond the scope of this guide. Refer to the appropriate IBM supplied manual for more details of job descriptions.

This parameter is a qualified name. Specify the name of the job description that is to be used. If required the library in which the job description resides can also be specified. If no library name is specified, library *LIBL is assumed which indicates the execution time library list of this job (ie: the job executing the SUBMIT command) should be searched to locate the job description.

If this parameter is not specified, default value QBATCH.*LIBL is assumed. This indicates that the library list of this job should be searched to locate a job description named QBATCH.

Special value *USRPRF indicates that the job description associated with the user profile under whose name the submitted job is to execute should be used. Normally submitted jobs execute under the user profile of the person who submitted the job.

Portability Considerations Refer to [IBM i Job Queue Emulation](#) .

JOBQ

Specifies the name (and optionally library) of the job queue onto which the batch job should be placed.

All batch jobs submitted on the IBM i must be placed onto a job queue. The details of what a job queue is and how it is used are beyond the scope of this guide. Refer to the appropriate IBM supplied manual for more details of job queues.

This parameter is a qualified name. Specify the name of the job queue that is to be used. If required the library in which the job queue resides can also be specified. If no library name is specified, library *LIBL is assumed which indicates the execution time library list of this job (ie: the job executing the SUBMIT command) should be searched to locate the job queue.

If this parameter is not specified, default value *JOBQ is assumed. This indicates that the job queue associated with the job description specified in the JOBQ parameter should be used.

Portability Considerations Refer to [IBM i Job Queue Emulation](#) .

OUTQ

Specifies the name (and optionally library) of the output queue onto which the batch job's output should be placed.

All batch jobs submitted on the IBM i must have an associated output queue. The details of what an output queue is and how it is used are beyond the scope of this guide. Refer to the appropriate IBM supplied manual for more details of output queues.

This parameter is a qualified name. Specify the name of the output queue that is to be used. If required the library in which the output queue resides can also be specified. If no library name is specified, library *LIBL is assumed which indicates the execution time library list of this job (ie: the job executing the SUBMIT command) should be searched to locate the output queue.

If this parameter is not specified, default value *JOBQ is assumed. This indicates that the output queue associated with the job description specified in the JOBQ parameter should be used.

Refer to the appropriate IBM supplied manual for more details of how the special parameter values *CURRENT, *USRPRF and *DEV can be used.

**Portability
Considerations**

Refer to the [SET_SESSION_VALUE](#) Built-In Function.

7.91.2 SUBMIT Comments / Warnings

Using Fields (i.e: Variables) in the Submit Command

The SUBMIT command parameters PGM, PROCESS, FUNCTION, JOB, JOBQ, JOBQ and OUTQ can all be specified as either alphanumeric literals (for example, **GLR001**) or as LANSAs field names (for example, **#PGMNAME**).

This allows a great deal of flexibility to be coded into one SUBMIT command. Consider the following RDML program:

```
REQUEST FIELDS(#OPTION)

CASE OF_FIELD(#OPTION)
  WHEN VALUE_IS('= 1')
    CHANGE FIELD(#FUNCTION) TO(PRINT)
  WHEN VALUE_IS('= 2')
    CHANGE FIELD(#FUNCTION) TO(BATCH)
  WHEN VALUE_IS('= 3')
    CHANGE FIELD(#FUNCTION) TO(PURGE)
  WHEN VALUE_IS('= 4')
    CHANGE FIELD(#FUNCTION) TO(BACKUP)
ENDCASE

SUBMIT PROCESS(ORDERS) FUNCTION(#FUNCTION) JOB(#FUNCTION)
```

This program allows the user to nominate an option of 1,2,3 or 4 and submit functions PRINT, BATCH, PURGE or BACKUP respectively (all of which belong to process ORDERS). The name of the job submitted will be the same as the function.

Submitting a Process or Function

- When submitting a process/function the parameters must exactly match those required by the process in number passed and type (ie: numeric or alpha).
- The parameters passed to the batch process or function can be a field name, an alphanumeric literal, a numeric literal, a system variable or a process parameter.
- Note that when numeric parameters are defined for a process they are **always packed decimal**. Thus any numeric parameters passed to LANSAs processes should also be packed decimal with the same length and number of decimal

positions. Failure to observe this rule may result in unpredictable results.

Calling a User Program

- The parameters passed to the batch program can be a field name, an alphanumeric literal, a numeric literal, a system variable or a process parameter.
- If special value *LIBL was nominated as the library containing the program to be invoked in batch then the program should be in one of the libraries specified in the INLLIBL (initial library list) parameter of the job description nominated in the JOBDD (job description) parameter of this command.

Submitting a Form

- When submitting a form, the value *FORM must be supplied to the SUBMIT command's FUNCTION parameter in a field. For example,
CHANGE FIELD(#FORM) TO ("*FORM")
SUBMIT PROCESS(ORDERFRM) FUNCTION (#FORM)

7.91.3 SUBMIT Examples

Example 1: Submit a call to a program named INVOICE in library PRODLIB and pass two parameters, invoice number and inquiry date as literals.

```
SUBMIT PGM(INVOICE.PRODLIB) PARM('INV123' '010187')
```

Example 2: Submit a call to program PUTBATCH passing the fields #BATCH, #ORDER and the current date (which is obtained from system variable *DATE) as parameters. Use job description GLEDGER and job queue QNIGHT to submit the job.

```
SUBMIT PGM(PUTBATCH) PARM(#BATCH #ORDER *DATE) JOBD(GLI
```

Example 3: Submit a call to LANSAPROCESS process ORDERS and request that the first function in the process be executed.

```
SUBMIT PROCESS(ORDERS)
```

Example 4: Submit a call to LANSAPROCESS process ORDERS2 and request that the function PURGE be executed. Exchange the order number and batch number with the function.

```
SUBMIT PROCESS(ORDERS2) FUNCTION(PURGE) EXCHANGE(#ORD
```

Example 5: Write a function called PRNTBCH (belonging to process GLPROC01) that when invoked interactively asks the user to specify the print criteria, submits **itself** to batch, and when invoked in batch uses the print criteria to produce the required report for the user. This example also illustrates the use of expandable groups to simplify field lists.

```
GROUP_BY NAME(#XG_EXCH) FIELDS(#GLNUMB #BATCH #MINCRE
IF COND(*JOBMODE = I')
    REQUEST FIELDS(#XG_EXCH)
    SUBMIT PROCESS(GLPROC01) FUNCTION(PRNTBCH) EXCHAN
ELSE
    SELECT FIELDS(#CREDIT #DEBIT) FROM_FILE(GLMASTV3) W.
    UPRINT FIELDS(#GLNUMB #BATCH #CREDIT #DEBIT)
    ENDSELECT
    ENDPRINT
ENDIF
```

When run interactively (ie: *JOBMODE = I) , this function requests that the user input a general ledger number, a batch number and a minimum credit amount. It then submits itself to batch, exchanging the general ledger number, batch number and minimum credit amount that were input by the user.

When invoked in **batch**, this function selects only records that match the user's request from logical file GLMASTV3 and prints them. The initial value of fields #GLNUMB, #BATCH and #MINCREDIT are established from the exchange list that was passed to the batch version of this function from the interactive version.

Example 6: Submit a form called ORDERFRM

```
CHANGE FIELD(#FORM) TO ("*FORM")  
SUBMIT PROCESS(ORDERFRM) FUNCTION (#FORM)
```

7.92 SUBROUTINE

The SUBROUTINE command is used to define the start of a subroutine and optionally nominate parameters which must be passed to it.

Portability Considerations Subroutines that are nested inside one another are not supported in the current release of Visual LANSA. This is a very rarely used coding technique and thus unlikely to cause any problems. In the event of problems simply unnest the subroutine(s) involved and recompile.

Also See

[7.92.1 SUBROUTINE Parameters](#)

[7.92.2 SUBROUTINE Comments / Warnings](#)

[7.92.3 SUBROUTINE Examples - Part 1](#)

[7.92.4 SUBROUTINE Examples - Part 2](#)

[7.38 ENDROUTINE](#)

[7.45 EXECUTE](#)

[EVTROUTINE](#)

[MTHROUTINE](#)

[PTYROUTINE](#)

Required

```
SUBROUTINE --- NAME ----- subroutine name -----  
----->
```

Optional

```
>-- PARMs ----- field name -- *BOTH -----|  
      |           *RECEIVED |  
      |           *RETURNED |  
      |           |  
      ----- 50 max -----
```

7.92.1 SUBROUTINE Parameters

NAME

PARMS

NAME

Specifies the name of the subroutine that is to be executed. The name used must be unique within the function.

PARMS

Optionally defines a list of parameters that must be passed to the subroutine by any EXECUTE command that uses it. All parameters must be defined as fields either in the LANSA data dictionary or in the function with a DEFINE command.

Following each field in the parameter list is an optional value that indicates whether the parameter is to be received or returned by the subroutine (or both).

*BOTH, which is the default value, indicates that the parameter is to be received from the value in the callers WITH_PARMS list, and returned into the callers WITH_PARMS value (if it is a field name).

*RECEIVED indicates that the parameter is to be received from the value in the caller's WITH_PARMS list, **but not returned** into the caller's WITH_PARMS value.

*RETURNED indicates that the parameter is **not to be received** from the value in the caller's WITH_PARMS list, but only returned into the caller's WITH_PARMS value (if it is a field name).

When executing a subroutine, the parameters specified in the WITH_PARMS parameter of the EXECUTE command must exactly match in number and type the parameters defined in the PARMS parameter of the associated SUBROUTINE command.

7.92.2 SUBROUTINE Comments / Warnings

The parameters **passed to** a subroutine by an EXECUTE command can be a field name, an alphanumeric literal, a numeric literal, a system variable or a process parameter.

The parameters **defined in** a subroutine (ie: in the PARMs parameter) must be field names. All fields used in the PARMs parameter must be defined in the LANSA data dictionary or within the function by a DEFINE command.

The values specified in the WITH_PARMs parameter of an EXECUTE command are mapped into the fields specified in the PARMs parameter of the associated SUBROUTINE command just prior to executing the subroutine when they have the attribute *BOTH or *RECEIVED.

When the subroutine has completed execution the fields nominated in the PARMs parameter of the SUBROUTINE command are conditionally mapped back into the values specified in the WITH_PARMs parameter of the EXECUTE command. The condition is that fields are **not mapped back** when the WITH_PARMs value is a literal value (alphanumeric or numeric), a system variable, a process parameter or the parameter has the attribute *RECEIVED.

The default mapping value for parameters is *BOTH. However, it is worth taking the time to specify *RECEIVED or *RETURNED as this will reduce the time taken for parameter mapping.

Subroutines can be coded **anywhere within a function** and even nested within one another like this:

```
SUBROUTINE NAME(SUB01)
.....
DISPLAY .. etc, etc

    SUBROUTINE NAME(SUB02)
    .....
    .....
    .....
    ENDROUTINE (subroutine SUB02)

CHANGE .. etc,etc
.....
```

```

SUBROUTINE NAME(SUB03)
.....
.....
.....
ENDROUTINE (subroutine SUB03)
.....
GOTO .....
.....
.....
ENDROUTINE (subroutine SUB01)

```

However, most programmers prefer to program subroutines at the end of the function, without nesting them inside one another.

There are very slight performance benefits in coding subroutines in decreasing order of use. If the most heavily used subroutines are coded first, then the operating system will have less work to do when you request that a subroutine be executed.

In the example above the command "following" (ie: executed after) the DISPLAY command is the CHANGE command even though subroutine SUB02 is between the 2 commands. Positioning SUB02 this way **does not** cause it be executed. The only way to execute SUB02 is via an EXECUTE command.

- The ability to code subroutines anywhere within a function and nest subroutines within one another **does not imply** any "scope" of the subroutines or the fields that they declare. The "scope" or "scoping" facility is a feature of some computer languages such as PL/1 and is not available in LANSA.

7.92.3 SUBROUTINE Examples - Part 1

Executing a SUBROUTINE

Executing a SUBROUTINE with parameters

Executing a SUBROUTINE with numeric literal as parameters

Executing a SUBROUTINE with alphanumeric literals as parameters

Executing a SUBROUTINE with system variables as parameters

Using SUBROUTINE to reduce coding

Using SUBROUTINE to print employee details

Techniques for documenting SUBROUTINES using the BBUSE template

Recursion

Subroutine variables are not locally scoped

Emulating local scoping by using a naming standard

Techniques for saving and restoring globally scoped variables

7.92.4 SUBROUTINE Examples - Part 2

Executing a SUBROUTINE

This is an example of how to execute a subroutine without passing any parameters:

```
EXECUTE SUBROUTINE(SUB1)
SUBROUTINE NAME(SUB1)
*      <<Logic>>
ENDROUTINE
```

Executing a SUBROUTINE with parameters

To pass parameters to a subroutine, use the WITH_PARMS() parameter in the EXECUTE command. You must make sure the EXECUTE command passes the same number of fields or values (WITH_PARMS) as the SUBROUTINE is expecting:

```
EXECUTE SUBROUTINE(SUB1) WITH_PARMS(#EMPNO)
EXECUTE SUBROUTINE(SUB2) WITH_PARMS(#GIVENAME #SURNAME)
EXECUTE SUBROUTINE(SUB3) WITH_PARMS(#SALARY #TOTAL)
SUBROUTINE NAME(SUB1) PARMS((#EMP1 *RETURNED))
CHANGE FIELD(#EMP1) TO(A0088)
ENDROUTINE
```

```

SUBROUTINE NAME(SUB2) PARMS((#NAME1 *RETURNED) (#NAME2
CHANGE  FIELD(#NAME1) TO(JOHN)
CHANGE  FIELD(#NAME2) TO(COOK)
ENDROUTINE
SUBROUTINE NAME(SUB3) PARMS((#WAGES *RECEIVED) (#SUM *R
CHANGE  FIELD(#WAGES) TO(230000)
CHANGE  FIELD(#SUM) TO('#WAGES * 1.1')
ENDROUTINE

```

In this example, any field specified with *RETURNED, like #EMP1, will be mapped back to #EMPNO when the subroutine completes.

If a function is coded with this logic then the final result for each subroutine for fields #EMPNO, #GIVENAME, #SURNAME and #TOTAL will be:

```

#EMPNO = A0080
#GIVENAME = John
#SURNAME = COOK
#TOTAL = 253,000.00

```

Executing a SUBROUTINE with numeric literal as parameters

In this example, subroutine A receives two numeric values, one passed in field #STD_NUM and a second one as a numeric literal (0.75). The subroutine makes a calculation and returns a value in the field #DISCOUNT.

```

DEFINE  FIELD(#DISCOUNT) TYPE(*DEC) LENGTH(10) DECIMALS(2)
DEFINE  FIELD(#Q) REFFLD(#STD_NUM)
DEFINE  FIELD(#N) TYPE(*DEC) LENGTH(3) DECIMALS(2)
DEFINE  FIELD(#D) TYPE(*DEC) LENGTH(10) DECIMALS(2) EDIT_C
BEGIN_LOOP
REQUEST  FIELDS(#STD_NUM)
EXECUTE  SUBROUTINE(A) WITH_PARMS(#STD_NUM 0.75 #DISCOU
DISPLAY  FIELDS(#DISCOUNT)
CHANGE  FIELD(#DISCOUNT) TO(#ZEROS)
END_LOOP
SUBROUTINE NAME(A) PARMS((#Q *RECEIVED) (#N *RECEIVED) (#I
CHANGE  FIELD(#D) TO('#Q * #N')
ENDROUTINE

```

A problem can occur if fields for *RECEIVED or *RETURNED parameters in a subroutine are not defined in the same field format (e.g. Length, type) with fields or literals that are used in the EXECUTE command as in this example:

```
DEFINE FIELD(#DISCOUNT) TYPE(*DEC) LENGTH(10) DECIMALS(2)
DEFINE FIELD(#Q) REFFLD(#STD_NUM)
DEFINE FIELD(#N) REFFLD(#STD_NUM)
DEFINE FIELD(#D) TYPE(*DEC) LENGTH(10) DECIMALS(2) EDIT_C
BEGIN_LOOP
EXECUTE SUBROUTINE(A) WITH_PARMS(1234 0.75 #DISCOUNT)
DISPLAY FIELDS(#DISCOUNT)
CHANGE FIELD(#DISCOUNT) TO(#ZEROS)
END_LOOP
SUBROUTINE NAME(A) PARMS((#Q *RECEIVED) (#N *RECEIVED) (#D *RECEIVED))
CHANGE FIELD(#D) TO('#Q * #N')
ENDROUTINE
```

Field #N is defined as packed 7,0

The literal 0.75 is passed to the subroutine, but this does not match the number of decimals that the subroutine is expecting for field #N.

As a result, the field #N received into the subroutine will incorrectly contain the value zero.

To work correctly, field #N should be defined as follows:

```
DEFINE FIELD(#N) TYPE(*DEC) LENGTH(3) DECIMALS(2)
```

Executing a SUBROUTINE with alphanumeric literals as parameters

In this example, subroutine STDNAME concatenates two strings from fields #STRING1 and #STRING2 into field #TEXT, which is mapped back (*RETURNED) into #CTEXT:

```
DEFINE FIELD(#OPTION) TYPE(*CHAR) LENGTH(1)
DEFINE FIELD(#CTEXT) TYPE(*CHAR) LENGTH(30) LABEL('Text')
BEGIN_LOOP
REQUEST FIELDS(#OPTION)
CASE OF_FIELD(#OPTION)
WHEN VALUE_IS('= C')
EXECUTE SUBROUTINE(STDNAME) WITH_PARMS(WILSON COOK:
DISPLAY FIELDS(#CTEXT)
OTHERWISE
```

```

MESSAGE  MSGTXT('Not a valid option')
ENDCASE
END_LOOP
SUBROUTINE NAME(STDNAME) PARMS((#STRING1 *RECEIVED) (#S
DEFINE   FIELD(#STRING1) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#STRING2) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#TEXT) TYPE(*CHAR) LENGTH(30)
USE      BUILTIN(CONCAT) WITH_ARGS(#STRING1 #STRING2) TO_G
ENDROUTINE

```

Executing a SUBROUTINE with system variables as parameters

It is also possible to pass system variables as parameters. In this example the system variable *FUNCTION:

```

DEFINE   FIELD(#TEMP) REFFLD(#FUNCTION)
DEFINE   FIELD(#TEMP2) REFFLD(#STD_TEXT)
DEFINE   FIELD(#TEXT) TYPE(*CHAR) LENGTH(20)
BEGIN_LOOP
EXECUTE  SUBROUTINE(VARIABLE) WITH_PARM(*FUNCTION #S
DISPLAY  FIELDS(#STD_TEXT)
MESSAGE  MSGTXT('Not a valid option')
END_LOOP
SUBROUTINE NAME(VARIABLE) PARMS((#TEMP *RECEIVED) (#TEM
CHANGE   FIELD(#TEXT) TO('Function name:')
USE      BUILTIN(CONCAT) WITH_ARGS(#TEXT #TEMP) TO_GET(#TE
ENDROUTINE

```

Using SUBROUTINE to reduce coding

SUBROUTINEs are a very useful way to reduce the amount of RDML code and at the same time make it simpler and easier to understand. Consider this case where the CHANGE command is used to add values to a working list:

```

CHANGE   FIELD(#EMPNO) TO(A0090)
CHANGE   FIELD(#NAME) TO('Fred')
CHANGE   FIELD(#SALARY) TO(23456.78)
ADD_ENTRY TO_LIST(#LIST)
CHANGE   FIELD(#EMPNO) TO(A0070)
CHANGE   FIELD(#NAME) TO('Mary')
CHANGE   FIELD(#SALARY) TO(43456.78)

```

```

ADD_ENTRY TO_LIST(#LIST)
CHANGE  FIELD(#EMPNO) TO(A0072)
CHANGE  FIELD(#NAME) TO('William')
CHANGE  FIELD(#SALARY) TO(33456.78)
ADD_ENTRY TO_LIST(#LIST)

```

By changing the code to use a subroutine which receives the values we want to add to the working list, the function becomes neater and the code more structured:

```

EXECUTE  SUBROUTINE(ADDTOLIST) WITH_PARM(A0090 'Fred' 23
EXECUTE  SUBROUTINE(ADDTOLIST) WITH_PARM(A0070 'Mary' 4
EXECUTE  SUBROUTINE(ADDTOLIST) WITH_PARM(A0072 'William
SUBROUTINE NAME(ADDTOLIST) PARM((#EMPNO *RECEIVED) (#N
ADD_ENTRY TO_LIST(#LIST)
ENDROUTINE

```

Using SUBROUTINE to print employee details

Similar to other examples, in this one we use a subroutine to execute a PRINT of employee details selected from PSLMST:

Using a subroutine to print a line is a useful method for handling those situations where it is necessary to print the same line in several different parts of the same LANSA function.

```

DEFINE  FIELD(#D1) REFFLD(#EMPNO)
DEFINE  FIELD(#D2) REFFLD(#SURNAME)
DEFINE  FIELD(#D3) REFFLD(#GIVENAME)
DEFINE  FIELD(#OPTION) TYPE(*CHAR) LENGTH(2)
DEF_LIST  NAME(#LIST1) FIELDS(#EMPNO #SURNAME #GIVENAME
BEGIN_LOOP
REQUEST  FIELDS(#OPTION)
CASE    OF_FIELD(#OPTION)
WHEN    VALUE_IS(= A')
SELECT  FIELDS(#EMPNO #SURNAME #GIVENAME) FROM_FILE(PS
ADD_ENTRY TO_LIST(#LIST1)
EXECUTE  SUBROUTINE(PRINT) WITH_PARM(#EMPNO #SURNAM
ENDSELECT
OTHERWISE
CALL    PROCESS(MYPROC)

```

```
ENDCASE
DISPLAY  FIELDS(#EMPNO #SURNAME #GIVENAME) BROWSELIST
END_LOOP
SUBROUTINE NAME(PRINT) PARMS((#D1 *RECEIVED) (#D2 *RECEIV
DEF_LINE  NAME(#NAME) FIELDS(#D1 #D2 #D3)
PRINT    LINE(#NAME)
ENDROUTINE
```

7.92.4 SUBROUTINE Examples - Part 2

Techniques for documenting SUBROUTINES using the BBUSE template
Recursion

Subroutine variables are not locally scoped

Emulating local scoping by using a naming standard

Techniques for saving and restoring globally scoped variables

7.92.3 SUBROUTINE Examples - Part 1

Techniques for documenting SUBROUTINES using the BBUSE template

When using the SUBROUTINE command it is often a good idea to use the LANSA template (for both Visual LANSA and LANSA for IBM i) called BBSUB.

This template will provide the basic coding layout for subroutine like this:

```
*=====
*Subroutine ....:
*Description....:
*=====
SUBROUTINE NAME(SUB1)
ENDROUTINE
```

If a parameter is required for the subroutine then this template will provide the basic coding layout automatically like this:

```
*=====
*Subroutine ....:
*Description....:
*Parameters ....: Name Type Len Description
*-----
*#XXXXXX   XXX 99,9 XXXXXXXXXXXXXXXXXXXXXXXX
*=====
SUBROUTINE NAME(SUB1) PARMS(#XXXX)
ENDROUTINE
```

For example subroutine named SUB1 has parameters #EMPNO, #SURNAME and #GIVENAME, these parameters can then be commented in the basic layout created by the BBUSE template to make the subroutine more understandable and easier to implement in the future.

Hence the layout can contain information about fields used as parameters used in the subroutine like this:

```
*=====
*Subroutine....:SUB1
*Description....: To retrieve an employee record from file PSLMST
*Parameters ....: #EMPNO, #SURNAME and #GIVENAME
*Name      Type   Len   Description
*-----
*#EMPNO1   A      5     Employee number
*#NAME1    A      20    Surname
*#NAME2    A      20    Givename
*=====
SUBROUTINE
NAME(SUB1) PARMS((#EMPNO1 *RETURNED) (#SURNAME *RETURNED) (#GIVENAME *RETURNED))
ENDROUTINE
```

Recursion

You should avoid recursively invoking **SUBROUTINEs**, either directly or indirectly.

Here **SUBROUTINE SUB_A** is invoked recursively by itself:

```
SUBROUTINE SUB_A
<< ETC >>
EXECUTE SUB_A
<< ETC >>
ENDROUTINE
```

Within Visual LANSA, this example will produce a fatal error, while in LANSA for i, it simply will fail to compile.

Here **SUBROUTINE SUB_A** is invoked recursively by **SUBROUTINE SUB_B**:

```
SUBROUTINE SUB_A
<< ETC >>
EXECUTE SUB_B
<< ETC >>
ENDROUTINE
```

```

SUBROUTINE SUB_B
<< ETC >>
EXECUTE SUB_A
<< ETC >>
ENDROUTINE

```

Once again within "Visual LANSA", this example will produce a fatal error, but in LANSA for i, it will end up in a recursive loop which will have to be ended manually.

Unlike subroutines MTHROUTINES (Method routines) in RDMLX are allowed to be recursive, so this factorial calculator should function correctly:

```

Mthroutine Factorial
Define_Map *input #Std_Num #OfNumber
Define_Map *output #Std_Num #ReturnResult
If      '#OfNumber.Value = 1'
Set     #ReturnResult Value(1)
Else
Change  #Std_Num '#OfNumber.Value - 1'
Invoke  #Com_Owner.Factorial OfNumber(#Std_Num) ReturnResult(#ReturnResult)
Change  #Std_NumL '#OfNumber.Value * #ReturnResult.Value'
Set     #ReturnResult Value(#Std_Num)
Endif
Endroutine

```

So **Invoke #Com_Owner.factorial ofNumber(4) ReturnResult(#Std_Num)** should return a result of **4 * 3 * 2 * 1 = 24**.

Subroutine variables are not locally scoped

Often the arguments received and returned by subroutines are defined within the subroutine like this:

```

SUBROUTINE NAME(A) PARMS(#A #B #C)
DEFINE  FIELD(#A) REFFLD(#SALARY)
DEFINE  FIELD(#B) REFFLD(#PERCENT)
DEFINE  FIELD(#C) REFFLD(#SALARY)

```

In RDML, such field definitions are simply a convention and are not locally

scoped. The fields are globally scoped within the RDML function (i.e: accessible to all the code in the function).

So in this example in this code, SUBROUTINE SUB_A will return the value 42.45, not 17.72:

```
FUNCTION OPTIONS(*DIRECT)
Define Field(#newsal) Reffld(#salary)
Execute Subroutine(SUB_A) With_Parms(#newsal)
Display Fields(#newsal)
```

```
Subroutine SUB_A ((#A *returned))
Define #A reffld(#salary)
Change #A 17.72
Execute SUB_B
Endroutine
```

```
Subroutine SUB_B
Change #A 42.45
Endroutine
```

This happens because #A is globally scoped. So when you reference #A in your code it is always the same instance of #A.

In RDMLX though, the EVTROUTINES, MTHROUTINES and PTYROUTINES do support local scoping.

If you coded the previous SUB_A and SUB_B subroutines as methods like this:

```
Function Options(*Direct)
Begin_Com Role(*Extends #Prim_Form)
Define_Com Class(#Salary.Visual) Name(#Salary) DisplayPosition(1) Height(
```

```
Evtoutine handling(#com_owner.Initialize)
Set #com_owner caption(*component_desc)
Invoke #com_owner.SUB_A A(#Salary)
Endroutine
```

```
Mthroutine SUB_A
Define_Map *output #Salary #A
Set #A Value(17.72)
Invoke #com_owner.SUB_B
```

Endroutine

Mthroutine SUB_B

Define_Com Class(#Salary) Name(#A)

Set #A Value(42.45)

Endroutine

End_Com

The method SUB_A would return 17.72.

This happens because there are two locally scoped #As defined.

One in method SUB_A and another method SUB_B.

If however you defined your code like this:

Define_Com Class(#Salary) Name(#A)

Mthroutine SUB_A

Define_Map *output #Salary #ReturnValue

Set #A Value(17.72)

Invoke #com_owner.SUB_B

Set #ReturnValue Value(#A)

Endroutine

Mthroutine SUB_B

Set #A Value(42.45)

Endroutine

Then method SUB_A would again return 42.45, because #A is a globally scoped component, so SUB_A and SUB_B are both referring to the same #A.

Emulating local scoping by using a naming standard

Even though locally scoped variables are not supported in subroutines you can (if required) emulate them by using a simple naming standard.

For example, each subroutine ensures that its arguments and variables are uniquely defined:

```
FUNCTION OPTIONS(*DIRECT)
```

```
DEFINE FIELD(#PERCENT) TYPE(*DEC) LENGTH(4) DECIMALS(1) I
```

```
REQUEST FIELDS(#EMPNO #PERCENT)
```

```
FETCH FIELDS(#SALARY) FROM_FILE(PSLMST) WITH_KEY(#EMPI
```

```

EXECUTE SUBROUTINE(SUB_A) WITH_PARMS(#SALARY #PERCENT
*
SUBROUTINE NAME(SUB_A) PARMS((#A_001 *Received)
(#B_001 *received)(#C_001 *Received))
DEFINE FIELD(#A_001) REFFLD(#SALARY)
DEFINE FIELD(#B_001) REFFLD(#PERCENT)
DEFINE FIELD(#C_001) REFFLD(#EMPNO)
CHANGE FIELD(#A_001) TO('#A_001 * #B_001')
DISPLAY FIELDS(#C_001 #A_001)
EXECUTE SUB_B (#A_001 #B_001 #C_001)
ENDROUTINE
*
SUBROUTINE NAME(SUB_B) PARMS((#A_002 *Received)
(#B_002 *received)(#C_002 *Received))
DEFINE FIELD(#A_002) REFFLD(#SALARY)
DEFINE FIELD(#B_002) REFFLD(#PERCENT)
DEFINE FIELD(#C_002) REFFLD(#EMPNO)
CHANGE FIELD(#A_002) TO('#A_002 - 500')
DISPLAY FIELDS(#C_002 #A_002)
EXECUTE SUBROUTINE(SUB_C) WITH_PARMS(#A_002 #C_002)
ENDROUTINE
*
SUBROUTINE NAME(SUB_C) PARMS((#A_003 *received)
(#C_003 *Received))
DEFINE FIELD(#A_003) REFFLD(#PERCENT)
DEFINE FIELD(#C_003) REFFLD(#EMPNO)
CHANGE FIELD(#A_003) TO('#A_003 - 100')
DISPLAY FIELDS(#EMPNO #A_003)
ENDROUTINE

```

If you code Execute SUB_A (#Salary #Percent #Empno), you are sure that these values being passed between the various subroutines will not inadvertently interfere with the globally scoped values of #Salary #Percent #Empno

Techniques for saving and restoring globally scoped variables

The previous example provides a simple technique for emulating locally scoped variables.

Sometimes when you are writing a subroutine you cannot avoid overwriting a globally scoped variable (e.g.: you have to fetch a field from a file).

Equally when you are maintaining a subroutine you cannot always be sure what other use is already being made of globally scoped variables elsewhere in the program without a detailed examination.

In these situations people typically use a simple save/restore technique to ensure that the globally scoped variable(s) remain unchanged by the execution of the subroutine.

For example, imagine you had to construct subroutine named SUB_A that needed to fetch the department (#DEPARTMENT) in which an employee worked in its logic:

```
Subroutine Name(SUB_A) Parms((#A_001 *received))  
Define #A_001 Reffld(#Empno)  
*<<etc>>  
Fetch (#Deptment) from_file(pslmst) with_key(#A_001)  
*<< etc >>  
Endroutine
```

Now imagine that field #DEPARTMENT was already used in several places in the program.

To ensure that you are not upsetting the value of field #DEPARTMENT in your new subroutine you would probably code this:

```
Subroutine Name(SUB_A) Parms((#A_001 *received))  
Define #A_001 Reffld(#Empno)  
Define #Save_001 Reffld(#Deptment)  
*<< etc >>  
Change #Save_001 #Deptment  
*<< etc >>  
Fetch (#Deptment) from_file(pslmst) with_key(#A_001)  
*<< etc >>  
Change #Deptment #Save_001  
*<< etc >>  
Endroutine
```

This ensures that the value of #DEPARTMENT is unchanged by the execution of your subroutine.

Now imagine that you also have to reference #SECTION, #SURNAME and #STARTDTE.

Your subroutine now looks like this:

```

Subroutine Name(SUB_A) Parms((#A_001 *received))
Define #A_001 Reffld(#Empno)
Define #SavA_001 Reffld(#Deptment)
Define #SavB_001 Reffld(#Section)
Define #SavC_001 Reffld(#Surname)
Define #SavD_001 Reffld(#Startdte)
*<< etc >>
Change #SavA_001 #Deptment
Change #SavB_001 #Section
Change #SavC_001 #Surname
Change #SavD_001 #Startdte
*<< etc >>
Fetch (#Deptment) from_file(pslmst) with_key(#A_001)
*<< etc >>
Change #Deptment #SavA_001
Change #Section #SavB_001
Change #Surname #SavC_001
Change #Startdte #SavD_001
*<< etc >>
Endroutine

```

However, by using a simple working list you can achieve the same result in a more efficient, easier to read and more maintainable manner:

```

Subroutine Name(SUB_A) Parms((#A_001 *received))
Define #A_001 Reffld(#Empno)

```

Global fields that may have been overwritten by this subroutine

```

Def_List #Save_001 (#Deptment #Section #Surname #StartDte) Type(*Worki

```

Save the value of all globally defined fields that may be overwritten

```

Inz_List #Save_001 Num_Entrys(1)
*<< etc >>
Fetch (#Deptment #Section #StartDte #Surname) from_file(pslmst) with_ke
*<< etc >>

```

Restore the value of all globally defined fields that may have been overwritten

```

Get_Entry 1 #Save_001

```

Endroutine

7.93 SUBSTRING

The SUBSTRING command is used to copy a string from one field to another field.

The string copied from the first field can be all or only part of the field.

The string can be copied into all or only part of the result field.

Also See

[7.93.1 SUBSTRING Parameters](#)

[7.93.2 SUBSTRING Examples](#)

Required

```
SUBSTRING ---- FIELD ----- field name ---- 1 ----
- *END --->
           field name field name
           (start pos) (length)

>--- INTO_FIELD --- field name ---- 1 ----- *END --
-|
           field name field name
           (start pos) (length)
```

7.93.1 SUBSTRING Parameters

FIELD

INTO_FIELD

FIELD

Specifies the field from which the string is to be extracted and optionally the start position and length of the string.

The field nominated in this parameter can be type alphanumeric or type numeric. When specifying a specific start position and length for a packed numeric field remember that digit positions and lengths are used **not** byte positions and lengths.

If no start position is specified for the string then start position 1 is assumed.

If no length is specified for the string then *END is assumed which indicates that all of the string from the start position to the end of the field is to be used.

The start position and length values can be nominated as either a literal value (e.g.: 10) or as the name of a field that contains the value (e.g: #LENGTH).

The start and end positions specified are validated at execution time (since they may be variable). If an invalid start or end position is specified the function will abort with an error message indicating the cause of the failure.

Special notes for substringing an alpha field into a numeric field:

Note 1. The field should only contain the digits 0-9. Any other character, including a sign character ('+' or '-'), will give unpredictable results.

Note 2. Substringing is from left to right, therefore if a field containing '123.45' is substringed into position 1 of a signed (6,2) field (which is initially set to *ZERO), the value will be set to 1234.50.

Note 3. The length and start position for Unicode fields is specified in characters, where every character is a 2-byte unit.

INTO_FIELD

Specifies the field into which the string extracted from the field nominated in the FIELD parameter is to be placed.

The field nominated in this parameter can be type alphanumeric or type numeric. When specifying a specific start position and length for a packed numeric field remember that digit positions and lengths are used **not** byte positions and lengths.

If no start position is specified for the string then start position 1 is assumed.

If no length is specified for the string then *END is assumed which indicates that all of the string from the start position to the end of the field is to be used.

The start position and length values can be nominated as either a literal value (e.g.: 10) or as the name of a field that contains the value (e.g.: #LENGTH).

The start and end positions specified are validated at execution time (since they may be variable). If an invalid start or end position is specified the function will abort with an error message indicating the cause of the failure.

7.93.2 SUBSTRING Examples

Example 1: If field #A is alphanumeric length 10 and field #B is alphanumeric length 5 then the following table indicates what happens when various SUBSTRING commands are used:

#A Before Substr.	#B Before Substr.	Substring Command	#B After Substr.
ABCDEFGHIJ	XXXXX	FIELD(#A) INTO_FIELD(#B)	ABCDE
ABCDEFGHIJ	XXXXX	FIELD(#A 1 1) INTO_FIELD(#B)	A
ABCDEFGHIJ	XXXXX	FIELD(#A 1 1) INTO_FIELD(#B 1 1)	AXXXX
ABCDEFGHIJ	XXXXX	FIELD(#A 2 2) INTO_FIELD(#B)	BC
ABCDEFGHIJ	XXXXX	FIELD(#A 2 2) INTO_FIELD(#B 2)	XBC
ABCDEFGHIJ	XXXXX	FIELD(#A 2 2) INTO_FIELD(#B 2 2)	XBCXX
ABCDEFGHIJ	XXXXX	FIELD(#A 9 1) INTO_FIELD(#B)	I
ABCDEFGHIJ	XXXXX	FIELD(#A 9 2) INTO_FIELD(#B)	IJ
ABCDEFGHIJ	XXXXX	FIELD(#A 9 2) INTO_FIELD(#B 2)	XIJ
ABCDEFGHIJ	XXXXX	FIELD(#A 9 2) INTO_FIELD(#B 2 2)	XIJXX

Example 2: Use the SUBSTRING command to alter a numeric six date field called #DDMMYY from format DDMMYY to YYMMDD:

```
DEFINE FIELD(#WORK02) TYPE(*CHAR) LENGTH(2)
```

```
SUBSTRING FIELD(#DDMMYY 1 2) INTO_FIELD(#WORK02)
SUBSTRING FIELD(#DDMMYY 5 2) INTO_FIELD(#DDMMYY 1 2)
SUBSTRING FIELD(#WORK02) INTO_FIELD(#DDMMYY 5 2)
```

Example 3: The following RDML program stores up to 20 product numbers input by the user in one long string called #PRODUCTS, then prints them all when no more are entered:

```
DEFINE FIELD(#PRODUCTS) TYPE(*CHAR) LENGTH(200)
DEFINE FIELD(#ENTERED) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEFAULT(0)
DEFINE FIELD(#I) TYPE(*DEC) LENGTH(3) DECIMALS(0)
```

```
DUNTIL '(#PRODNO = *BLANKS) *OR (#ENTERED = 20)'
CHANGE #PRODNO *BLANKS
REQUEST FIELDS(#PRODNO)
IF '#PRODNO *NE *BLANKS'
CHANGE #ENTERED ('#ENTERED + 1')
CHANGE #I '((#ENTERED - 1) * 10) + 1'
SUBSTRING FIELD(#PRODNO) INTO_FIELD(#PRODUCTS #I 10)
ENDIF
ENDUNTIL
```

```
DOWHILE '#ENTERED *GT 0'
CHANGE #I '((#ENTERED - 1) * 10) + 1'
SUBSTRING FIELD(#PRODUCTS #I 10) INTO_FIELD(#PRODNO)
FETCH FIELDS(#DESCRIPT #PRICE #QUANTITY) FROM_FILE(PROD)
UPRINT FIELDS(#PRODNO #DESCRIPT #PRICE #QUANTITY)
CHANGE #ENTERED ('#ENTERED - 1')
ENDWHILE
```

7.94 TRANSFER

The TRANSFER command is used to transfer control from one function to another function. Optionally information may be exchanged with the other function. The TRANSFER command can also be used in WAM Components to transfer control to other WEBROUTINEs in the same WAM Component or another WAM Component.

Also See

[7.94.1 TRANSFER Parameters](#)

[7.94.2 TRANSFER Comments / Warnings](#)

[7.94.3 TRANSFER Examples](#)

Optional

```
TRANSFER ----- TOFUNCTION -- *NEXT -----
----->
        *LAST
        *EXIT
        *MENU
        *HELP
        *EOJ
        function name

>---- EXCHANGE ----- field name -----|
        | expandable group expression |
        |                               |
        ----- 100 max -----

>---- TOROUTINE ---- webroutine name -----
-->
        *SERVICE service name
        *EVALUATE field name

>---- ONENTRY ----- *MAP_NONE -----
-->
        *MAP_ALL
        *MAP_LOCAL
        *MAP_SHARED
```

7.94.1 TRANSFER Parameters

EXCHANGE

ONENTRY

TOFUNCTION

TOROUTINE

TOFUNCTION

Specifies the name of the function that is to receive control. The current function ends when this command is executed and the function nominated in this parameter receives control.

*NEXT, which is the default value, indicates that the function name input from the screen should receive control next. Usually this is the function nominated in the function control table as the default next function. However, dependent upon the menu selection style and the SHOW_NEXT parameter of the DISPLAY or REQUEST commands used, it may have been changed by the user. Refer to [Function Control Table](#) for information about the function control table before attempting to use this parameter.

*LAST indicates that control should be transferred to the function that was in use immediately prior to this one.

*EXIT indicates that control should be passed out of the LANSAs system to the application that invoked LANSAs. Using this option is identical to using the EXIT command.

*MENU indicates that control should be passed to the process's main menu. Using this option is identical to using the MENU command.

*HELP indicates that control should be passed to the online HELP facility. Using this option is similar to using the HELP function key.

*EOJ indicates that a batch function should end. Using this option is similar to using the *EXIT option in an interactive function.

If one of the previous values is not used then the name of another function that is defined within the same process must be specified.

EXCHANGE

Specifies the name of the field(s) to be exchanged or the name of a group that defines the field(s) to be exchanged. When an expandable expression is used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

Refer to [Field Groups and Expandable Groups](#) for more information of how field and group names can be specified. Refer to the [7.42 EXCHANGE](#) command for more information about how information is exchanged between functions.

The EXCHANGE parameter on this command is provided for convenience only. Using it is identical to using one or more EXCHANGE commands before the TRANSFER command. Thus:

```
TRANSFER TOFUNCTION(INPUT) EXCHANGE(#A #B #C #D)
```

is functionally identical to:

```
EXCHANGE FIELDS(#A #B #C #D)  
TRANSFER TOFUNCTION(INPUT)
```

which is functionally identical to:

```
EXCHANGE FIELDS(#A)  
EXCHANGE FIELDS(#B)  
EXCHANGE FIELDS(#C)  
EXCHANGE FIELDS(#D)  
TRANSFER TOFUNCTION(INPUT)
```

TOROUTINE

Specifies the name of a WEBROUTINE to transfer to. You can specify another WAM, in this case the WAM name followed by a WEBROUTINE name separated by a dot (for example #MyWAM.MyWebRtn). Unlike a CALL command, after a transfer control does not return to the WEBROUTINE making the transfer.

A Service Name can also be specified, if prefixed with the *SERVICE modifier. The value can also be provided from a field, if prefixed with the *EVALUATE modifier.

ONENTRY

Is valid when transferring to another WEBROUTINE only. For WEBROUTINE information, refer to [WEBROUTINE](#).

Used for mapping incoming fields and lists into the target WEBROUTINE. This property can be one of:

*MAP_NONE does not map any fields or lists.

*MAP_ALL maps all required fields and lists.

*MAP_LOCAL only fields and lists on WEBROUTINE's WEB_MAPs are mapped.

*MAP_SHARED only WAM level WEB_MAP fields and lists are mapped, not WEBROUTINE level.

The default value is *MAP_ALL.

7.94.2 TRANSFER Comments / Warnings

Information is exchanged between functions in an "exchange list". The format of the exchange list is something like this:

```
||N|T|L|D| V ||N|T|L|D| V | ..... |N|T|L|D| V ||
```

where:

N is the name of a field

T is the type of a field

L is the length of a field

D is the number of decimal positions

V is the variable length value of the field

Whenever a function is invoked the exchange list is searched. If a field is found in the exchange list with the same name as a field used in the function it is "mapped" into the function. After the search has been completed the exchange list is cleared, regardless of **whether or not** any fields were found in it and mapped into the function.

It can be seen that the exchange of information between functions is by **name, not by position** as with normal program parameters.

The "mapping" procedure mentioned above will automatically convert field types, lengths and decimal positions if the definition of the field in the exchange list is different from the definition of the field in the function.

The EXCHANGE command or the EXCHANGE parameter on the TRANSFER command are used to add a new entry into the EXCHANGE list.

The net length of the exchange list cannot exceed 2000 characters at any time or the EXCHANGE or TRANSFER command that is attempting to add information to the list will end abnormally.

When working with the function that receives the EXCHANGE information remember that the exchange of information takes place before the first RDML command in the function is executed.

If the EXCHANGE command or parameter does not appear to be working correctly it is probably because the first RDML command in the function sets the fields that have just been mapped from the exchange list to *DEFAULT or *NULL. This causes the EXCHANGE values to be lost / overwritten.

7.94.3 TRANSFER Examples

Example 1: Transfer control to the default next function. No information is to be exchanged:

```
TRANSFER
```

Example 2: Transfer control to a function named INPUT. No information is to be exchanged:

```
TRANSFER TOFUNCTION(INPUT)
```

Example 3: Transfer control to a function named INPUT. Exchange the values of fields #CUSTNO, #BATCH and #USER with it:

```
TRANSFER TOFUNCTION(INPUT) EXCHANGE(#CUSTNO #BATCH #U
```

Example 4: Transfer control to WEBROUTINE ORDER:

```
TRANSFER TOROUTINE(ORDER)
```

Values of any fields and lists specified FOR(*INPUT) on the ORDER WEBROUTINE will be passed to it.

Example 5: Transfer control to WEBROUTINE ORDER in ORDERS WAM:

```
TRANSFER TOROUTINE(#ORDERS.ORDER)
```

Values of any fields and lists specified FOR(*INPUT) on the ORDER WEBROUTINE will be passed to it.

Example 6: Provide the name of a WEBROUTINE to transfer control to, from a field:

```
#WEBRTN := 'ORDERS.ORDER'  
TRANSFER TOROUTINE(*EVALUATE #WEBRTN)
```

7.95 UPD_ENTRY

The UPD_ENTRY command is used to update an existing entry in a list.

The list may be a **browse** list (used for displaying information at a workstation) or a **working** list (used to store information within a program).

Before a list entry can be updated via the UPD_ENTRY command it must first have been selected (ie: retrieved) from the list in a SELECTLIST / ENDSELECT list processing loop or by a GET_ENTRY or LOC_ENTRY command.

Refer to the DEF_LIST command for more details of lists and list processing.

Also See

[7.95.1 UPD_ENTRY Parameters](#)

[7.95.2 UPD_ENTRY Comments / Warnings](#)

[7.95.3 UPD_ENTRY Examples](#)

Optional

```
UPD_ENTRY ---- IN_LIST ----- *FIRST -----
---->
      list name

>-- WITH_MODE ---- *CURRENT -----
---|
      *ADD
      *CHANGE
      *DELETE
      *DISPLAY
      *SAME
      field name
```

7.95.1 UPD_ENTRY Parameters

[IN_LIST](#)

[WITH_MODE](#)

IN_LIST

Specifies the name of the list in which the entry should be updated.

The default value of *FIRST specifies that the first list declared in the RDML program by a DEF_LIST (define list) command is the list to be used (which may be a browse or working list).

If a list name is used then the list name must be declared elsewhere in the RDML program by a DEF_LIST (define list) command.

WITH_MODE

Specifies the mode to be set for the entry being updated. This overrides the mode that has been set by the SET_MODE command (refer to the SET_MODE command).

The default is *CURRENT which uses the current mode that has been set by the SET_MODE command. Other allowable values are *ADD, *CHANGE, *DELETE, *DISPLAY and *SAME (leave list entry in same mode as it was when added to the list, i.e. if it was added to the list with a mode of *CHANGE then leave the entry in *CHANGE mode). A user field name may also be specified, and must be alphanumeric with a length of 3, and must contain one of the values "ADD", "CHG", "DLT" or "DIS".

7.95.2 UPD_ENTRY Comments / Warnings

UPD_ENTRY is a "**mode sensitive**" command when being used with a browse list. Refer to [RDML Screen Modes and Mode Sensitive Commands](#) for details.

IBM i and CPF operating system restrictions **prevent** logic like the following example from **ever working correctly** on two (or more) browse lists:

```
SELECTLIST NAMED(#LIST01)
<< process LIST01 entry >>
    SELECTLIST NAMED(#LIST02)
    << process LIST02 entry >>
    UPD_ENTRY IN_LIST(#LIST02)
    ENDSELECT
UPD_ENTRY IN_LIST(#LIST01)
ENDSELECT
```

The reason is that all browse lists belong to the same "file" (ie: display file) and therefore the update implicitly attempts to update the last "record" processed in the "file". If a program like this example was compiled, it would fail on the UPD_ENTRY command to #LIST01 with an error indicating an update was attempted "without a prior read".

In other words, you can only update a browse list entry if the last operation performed on **any** browse list was a **read** operation against the **browse list that is being updated** (ie: SELECTLIST or GET_ENTRY).

This restriction can usually be overcome by altering the point at which the update operation is performed like this:

```
SELECTLIST NAMED(#LIST01)
<< process LIST01 entry >>
=> UPD_ENTRY IN_LIST(#LIST01)
    SELECTLIST NAMED(#LIST02)
    << process LIST02 entry >>
    UPD_ENTRY IN_LIST(#LIST02)
    ENDSELECT
ENDSELECT
```

If a solution like this cannot be implemented, use a GET_ENTRY command immediately before the UPD_ENTRY command.

This restriction does **not** apply to working lists.

7.95.3 UPD_ENTRY Examples

Example 1: Define, initialize and accept input into a list from the workstation. Process and validate the input, then update the database:

```
DEF_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #Q
SET_MODE TO(*ADD)
CHANGE FIELD(#ORDERLINE) TO(*NULL)
INZ_LIST NAMED(#ORDERLINE) NUM_ENTRYYS(10)

DISPLAY BROWSELIST(#ORDERLINE)

BEGINCHECK
SELECTLIST NAMED(#ORDERLINE) GET_ENTRYYS(*NOTNULL)

* VALUECHECK --
* CONDCHECK | Various validation checks
* RANGECHECK | performed against each list entry
* FILECHECK --

UPD_ENTRY IN_LIST(#ORDERLINE)
ENDSELECT

ENDCHECK IF_ERROR(*LASTDIS)

SELECTLIST NAMED(#ORDERLINE) GET_ENTRYYS(*ALL)
INSERT FIELDS(#ORDERLINE) TO_FILE(ORDLIN)
ENDSELECT
```

7.96 UPDATE

The UPDATE command allows the field(s) in record(s) in a file to be updated.

Portability Considerations Refer to the parameters: [AUTOCOMMIT](#), [FIELDS](#) and [IN_FILE](#) .

Also See

[7.96.1 UPDATE Parameters](#)

[7.96.2 UPDATE Comments / Warnings](#)

[7.96.3 UPDATE Examples](#)

```

                                         Required
UPDATE ----- FIELDS -----
- field name field attributes --->
    |          |          ||
    |          --- 7 max ----- |
    |*ALL          |
    |*ALL_REAL      |
    |*ALL_VIRT      |
    |*EXCLUDING     |
    |*INCLUDING     |
    |expandable group |
    |                |
    |----- 1000 max for RDMLX-----|
    ----- 100 max for RDML -----

>-- IN_FILE ----- file name . *FIRST ----->

-----
                                         Optional
>-- WITH_KEY ----- key value ----->
    |expandable group expression|
    ----- 20 maximum -----

>-- IO_STATUS ----- *STATUS ----->
    field name

>-- IO_ERROR ----- *ABORT -----
```

>

**NEXT*
**RETURN*
command label

>-- *VAL_ERROR* ---- **LASTDIS* -----

>

**NEXT*
**RETURN*
command label

>-- *NOT_FOUND* ---- **NEXT* -----

->

**RETURN*
command label

>-- *ISSUE_MSG* ---- **NO* ----->
**YES*

>-- *WITH_RRN* ---- **NONE* -----

>

>-- *RETURN_RRN* --- **NONE* -----

->

>-- *CHECK_ONLY* --- **NO* -----

>

**YES*

>-- *AUTOCOMMIT* --- **FILEDEF* -----

---|

**YES*
**NO*

7.96.1 UPDATE Parameters

AUTOCOMMIT

CHECK_ONLY

FIELDS

IN_FILE

IO_ERROR

IO_STATUS

ISSUE_MSG

NOT_FOUND

RETURN_RRN

VAL_ERROR

WITH_KEY

WITH_RRN

FIELDS

Specifies either the field(s) that are to be updated in the file or the name of a group that specifies the field(s) to be updated.

An expandable group expression is allowed in this parameter. Refer to [Field Groups and Expandable Groups](#) for more details. The following special values can be used:

- *ALL, specifies that all fields from the currently active file be updated.
- *ALL_REAL, specifies that all real fields from the currently active file be updated.
- *ALL_VIRT, specifies that all virtual fields from the currently active file be updated.
- *EXCLUDING, specifies that fields following this special value must be excluded from the field list.
- *INCLUDING, specifies that fields following this special value must be included in the field list. This special value is only required after an *EXCLUDING entry has caused the field list to be in exclusion mode.

Note: When all fields are updated through a logical file maintained by OTHER, all the fields from the based-on physical file are included in the field list.

It is strongly recommended that the special values *ALL, *ALL_REAL or *ALL_VIRT in parameter FIELDS be used sparingly and only when strictly required. Updating fields which are not needed invalidates cross-reference details (shows fields which are not used in the function) and increases the Crude Entity Complexity Rating of the function pointlessly.

Portability Considerations On IBM i, if one or more LOB fields are to be updated, and the file is not under commitment control, if an I/O error occurs, it is possible that the non-LOB fields have been updated, but one or more LOB fields have not.

Refer also to [Commitment Control](#) in the *LANSA Application Design Guide*.

IN_FILE

Refer to [Specifying File Names in I/O commands](#).

WITH_KEY

Refer to [Specifying File Key Lists in I/O Commands](#).

Also refer to the comments/warnings section following for details of how using this parameter affects automatic "crossed update" checking.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

Refer to [I/O Command Return Codes Table](#) for I/O operation return codes.

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the

function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command. The purpose of *NEXT is to permit you to handle error messages in the RDML, and then ABORT, rather than use the default ABORT. (It is possible for processing to continue for LANSAs for i and Visual LANSAs, but this is NOT a recommended way to use LANSAs.)

ER returned from a database operation is a fatal error and LANSAs does not expect processing to continue. The IO Module is reset and further IO will be as if no previous IO on that file had occurred. Thus you must not make any presumptions as to the state of the file. For example, the last record read will not be set. A special case of an IO_ERROR is when a trigger function is coded to return ER in TRIG_RETC. The above description applies to this case as well. Therefore, LANSAs recommends that you do NOT use a return code of ER from a trigger function to cause anything but an ABORT or EXIT to occur before any further IO is performed.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

VAL_ERROR

Specifies the action to be taken if a validation error was detected by the command.

A validation error occurs when information that is to be added, updated or deleted from the file does not pass the FILE or DICTIONARY level validation checks associated with fields in the file.

If the default value *LASTDIS is used control will be passed back to the last display screen used. The field(s) that failed the associated validation checks will be displayed in reverse image and the cursor positioned to the first field in error on the screen.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

The *LASTDIS is valid even if there is no "last display" (such as in batch functions). In this case the function will abort with the appropriate error message(s).

When using *LASTDIS the "Last Display" must be at the same level as the database command (INSERT, UPDATE, DELETE, FETCH and SELECT). If they are at different levels e.g. the database command is specified in a SUBROUTINE, but the "Last Display" is a caller routine or the mainline, the function will abort with the appropriate error message(s).

The same does NOT apply to the use of event routines and method routines in Visual LANSA. In these cases, control will be returned to the calling routine. The fields will display in error with messages returned to the first status bar encountered in the parent chain of forms, or if none exist, the first form with a status bar encountered in the execution stack (for example, a reusable part that inherits from PRIM_OBJT).

NOT_FOUND

Specifies what is to happen if no record is found in the file to be updated.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

ISSUE_MSG

Specifies whether a "record not found" message is to be automatically issued or not.

The default value is *NO which indicates that no message should be issued.

The only other allowable value is *YES which indicates that a message should be automatically issued. The message will appear on line 22/24 of the next screen format presented to the user or on the job log of a batch job.

WITH_RRN

Specifies the name of a field that contains the relative record number (for relative record file processing) of the record which is to be updated.

The WITH_RRN parameter cannot be used with the WITH_KEY parameter.

Any field nominated in this parameter must be defined within the function or the LANSAs data dictionary and must be numeric.

Note: Using the WITH_RRN parameter to FETCH, DELETE or UPDATE records is faster than any other form of database access.

The actual database file being accessed is always the physical file, regardless of whether or not the file nominated in the command is a logical file. Thus logical file select/omit criteria are not used when accessing a logical file via the WITH_RRN parameter.

Refer also to:

- [7.96.2 UPDATE Comments / Warnings](#) for details of how using this parameter affects automatic "crossed update" checking.
- [Load Other File](#) in the *Visual LANSAs Developers Guide*.

RETURN_RRN

Specifies the name of a field in which the relative record number of the record just updated should be returned. The value returned in this field is not usable when the UPDATE command is updating multiple records in the file.

Any field nominated in this parameter must be defined within the function or the LANSAs data dictionary and must be numeric.

For further information refer also to [Load Other File](#) in the *Visual LANSAs Developers Guide*.

CHECK_ONLY

Indicates whether the I/O operation should actually be performed or only "simulated" to check whether all file and data dictionary level validation checks can be satisfied when it is actually performed.

*NO, which is the default value, indicates that the I/O operation should be performed in the normal manner.

*YES indicates that the I/O operation should be simulated to verify that all file and data dictionary level checks can be satisfied. The database file involved is not changed in any way when this option is used.

AUTOCOMMIT

This parameter was made redundant in LANSAs release 4.0 at program change level E5.

To use commitment control specify COMMIT and/or ROLLBACK commands in your application.

Generally only COMMIT commands are required.

For the implications of using commitment control on the IBM i, refer to [Commitment Control](#) in the *LANSAs for I User Guide*.

Portability If using Visual LANSAs, refer to [Commitment Control](#) in the
Considerations *LANSAs Application Design Guide*.

7.96.2 UPDATE Comments / Warnings

Understand UPDATE Command

The use of automatic "**crossed update**" checks by the UPDATE command should be clearly understood.

Consider the following flow of commands:

```
FETCH WITH_KEY( ) or WITH_RRN( )
DISPLAY
IF_MODE *CHANGE
UPDATE
ENDIF
```

Since the UPDATE command has **no** WITH_KEY or WITH_RRN parameter it is indicating that the **last record read** (by the FETCH command) should be updated.

In this situation, the "**crossed update window**" is in the interval between the time the record was FETCHed and the time that it is UPDATED. This could be **very long** if the user went and had a cup of coffee when the DISPLAY command was on their workstation.

This is a **correct and valid** use of the automatic "crossed update" checking facility. If the record was changed by another job/user between the FETCH and the UPDATE, then the UPDATE will generate a "crossed update error" (which should be handled just like any other type of validation error).

Now consider the following flow of commands:

```
FETCH WITH_KEY( ) or WITH_RRN( )
DISPLAY
IF_MODE *CHANGE
UPDATE WITH_KEY( ) or WITH_RRN( )
ENDIF
```

Since the UPDATE command **has** a WITH_KEY or WITH_RRN parameter it is indicating that a specific record (or group of records) should be **read** and **updated**.

This is a common coding mistake. Everybody knows that the WITH_KEY or WITH_RRN values on the UPDATE command should/would be the same as

those on the FETCH command. However, the RDML compiler cannot be sure that the values were not changed, so it is **forced** to (re)read the record before attempting the UPDATE.

In this situation, the "**crossed update window**" is in the interval between the time the record is (re)read by the UPDATE command and then updated by the UPDATE command. This interval is very short, and thus the "crossed update" check is effectively disabled.

This is **not** considered to be a **valid and correct** use of the UPDATE command in an **interactive program** like this because it effectively disables the automatic "crossed update" check.

No KEY

Where an UPDATE operation is issued with no WITH_KEY or WITH_RRN parameters specified the last record read from the file will be updated. Thus the following are equivalent operations:

```
CHANGE FIELD(#DATDUE) TO(*DATE)
UPDATE FIELDS(#DATDUE) IN_FILE(ORDHDR) WITH_KEY(#ORDNUM)
```

is functionally equivalent to:

```
FETCH FIELDS(#DATEDUE) FROM_FILE(ORDHDR) WITH_KEY(#ORDNUM)
CHANGE FIELD(#DATDUE) TO(*DATE)
UPDATE FIELDS(#DATDUE) IN_FILE(ORDHDR)
```

and:

```
CHANGE FIELD(#QUANTITY) TO(100)
UPDATE FIELDS(#QUANTITY) IN_FILE(ORDLIN) WITH_KEY(#ORDNUM)
```

is functionally equivalent to:

```
SELECT FIELDS(#QUANTITY) FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM)
CHANGE FIELD(#QUANTITY) TO(100)
UPDATE FIELDS(#QUANTITY) IN_FILE(ORDLIN)
ENDSELECT
```

Note that the last 2 examples change the #QUANTITY field of **all** order lines to

100. This is an example of multiple record updating (or "set at a time" updating).

Note 'UPDATE WITH_KEY' should not be used within a select loop or in a subroutine called from within a select loop.

SQL NULL

When an SQL Null field is updated into a table's database column, one of the following will occur:

- If the column does not have the NOT NULL constraint, the column is set to SQL Null.
- If the column does have the NOT NULL constraint, the update will fail. (This can only occur if the database definition of the column does not match the LANSAs definition of the field.)

7.96.3 UPDATE Examples

Example 1: Ask the user to input an order number and customer number from the workstation and update the associated order header record:

```
REQUEST FIELDS(#ORDNUM #CUSTNO)
UPDATE FIELDS(#ORDNUM #CUSTNO) IN_FILE(ORDHDR) WITH_KEY(#ORDNUM #CUSTNO)
```

Example 2: Ask the user to input an existing order number and a new order number at the workstation. Update the existing order header and **all** associated order lines to have the new order number:

```
DEFINE FIELD(#OLDORDNUM) REFFLD(#ORDNUM) LABEL('Existing ord num')
DEFINE FIELD(#NEWORDNUM) REFFLD(#ORDNUM) LABEL('New ord num')

REQUEST FIELDS(#OLDORDNUM #NEWORDNUM)

CHANGE FIELD(#ORDNUM) TO(#NEWORDNUM)
UPDATE FIELDS(#ORDNUM) IN_FILE(ORDHDR) WITH_KEY(#OLDORDNUM #NEWORDNUM)
UPDATE FIELDS(#ORDNUM) IN_FILE(ORDLIN) WITH_KEY(#OLDORDNUM #NEWORDNUM)
```

Example 3: Change the price (#PRICE) of all records in a nominated product category (#CAT) to \$100.00:

```
CHANGE FIELD(#PRICE) TO(100.00)
UPDATE FIELDS(#PRICE) IN_FILE(PROMSTV1) WITH_KEY(#CAT)
```

Example 4: Increase the price (#PRICE) of all records in a nominated product category (#CAT) by \$100.00:

```
SELECT FIELDS(#PRICE) FROM_FILE(PROMSTV1) WITH_KEY(#CAT)
CHANGE FIELD(#PRICE) TO('#PRICE + 100.00')
UPDATE FIELDS(#PRICE) IN_FILE(PROMSTV1)
ENDSELECT
```

Example 5: Reset all cost related fields in a nominated product category (#CAT):

```
GROUP_BY NAME(#XG_COST) FIELDS(#COST1 #COST2 #COST3 #COST4)
CHANGE FIELD(#XG_COST) TO(*DEFAULT)
UPDATE FIELDS(#XG_COST) IN_FILE(PROMSTV1) WITH_KEY(#CAT)
```


7.97 UPRINT

The UPRINT command is used to print fields onto a report.

The UPRINT command can be used to produce simple paginated listings with subtotals. Refer to the [RDML Field Attributes and their Use](#) for details of which field attributes can be used in with the UPRINT command.

The use of the UPRINT command is recommended only for very simple list style reports. For serious application reporting, multilingual reporting or bi-directional language reporting use **only** the PRINT command.

For producing complex reports or reports with specific layout requirements refer to the PRINT command.

Refer also to the ENDPRINT command, which is used to close (end) a report produced by using the UPRINT command.

Portability Considerations This command is not supported in Visual LANSA and is not expect to be in future releases. A build warning will be generated if used and an error will occur at execution time. Code using this facility can be made conditional so that it is not executed in this environment.

Also See

[7.97.1 UPRINT Parameters](#)

[7.97.2 UPRINT Examples](#)

Required

```
UPRINT ----- FIELDS ----- field name field attributes -  
-->
```

```
      |           |           ||  
      |           --- 7 max ----- |  
----- 100 max -----
```

Optional

```
>-- TITLE ----- *NONE ----->  
      'report title'
```

>-- *REPORT_NUM* --- 1 ----->
 report number 1 -> 9

>-- *SPACE* ----- 1 ----->
 lines to space

>-- *WIDTH* ----- **DEFAULT* -----
>
 report width

>-- *COLUMN_SEP* --- 1 ----->
 column separation

>-- *START_COL* ---- 1 ----->
 start column number

>-- *IO_STATUS* ---- **STATUS* ----->
 field name

>-- *IO_ERROR* ---- **ABORT* -----|

7.97.1 UPRINT Parameters

COLUMN_SEP

FIELDS

IO_ERROR

IO_STATUS

REPORT_NUM

SPACE

START_COL

TITLE

7.98 USE

FIELDS

Specifies either the field(s) that are to be printed or the name of a group that specifies the field(s) to be printed.

TITLE

Specifies the title (if any) that is to be printed on the report.

*NONE, which is the default value indicates that no title is required.

If a title is required specify the title in quotes.

REPORT_NUM

Specifies the number of the report that is to be used to print the line. If no report number is specified report number 1 is assumed. The report number specified can be any number in the range 1 to 9. Up to 9 reports can be produced simultaneously.

SPACE

Specifies the number of lines to be spaced before printing the line. If no value is specified 1 is assumed. The value specified must be in the range 1 to 3.

WIDTH

Specifies the width (in characters) of the report. If no value is specified *DEFAULT is assumed which means that the system default report width will be used. Refer to other sections in this guide for more details of system default values. Otherwise specify a value in the range 1 to 198.

COLUMN_SEP

Specifies the number of spaces (in characters) that are to be left between columns in the report. If no value is specified 1 is assumed. Otherwise specify a value in the range 1 to 20.

START_COL

Specifies the column in which the first field is to be printed. If no value is specified 1 is assumed. Otherwise specify a value in the range 1 to 198.

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

Refer to [I/O Command Return Codes Table](#) for I/O operation return codes.

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

If the default value *ABORT is not used you must nominate a valid command label to which control should be passed if an I/O error occurs.

7.97.2 UPRINT Examples

Example 1: The following RDML program asked the user to input an order number and then prints details of the order and its associated order lines.

```
GROUP_BY NAME(#ORDERDET) FIELDS(#ORDNUM #CUSTNUM #  
  
L1: REQUEST FIELDS(#ORDNUM)  
FETCH FIELDS(#ORDERDET) FROM_FILE(ORDHDR) WITH_KEY  
  
SELECT FIELDS(#ORDERDET) FROM_FILE(ORDLIN) WITH_KEY(  
UPRINT FIELDS(#ORDERDET)  
ENDSELECT  
  
ENDPRINT
```

Example 2: If a file called ACCOUNT contains the following fields and data:

Company (#COMP)	Division (#DIV)	Department (#DEPT)	Expenditure (#EXPEND)	Revenue (#REVNU)
01	1	ADM	400	576
"	"	MKT	678	56
"	"	SAL	123	6784
"	2	ADM	46	52
"	"	SAL	978	456
"	3	ACC	456	678
"	"	SAL	123	679
02	1	ACC	843	400
"	"	MKT	23	0
"	"	SAL	876	10
"	2	ACC	0	43

and if the file is keyed by #COMP, #DIV and #DEPT, then the following RDML program will produce a paginated report with subtotals from this file:

```
GROUP_BY NAME(#ACCOUNTS) FIELDS((#COMP *TOTLEVEL1 *NE  
  
SELECT  FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT)  
UPRINT  FIELDS(#ACCOUNTS)  
ENDSELECT  
  
ENDPRINT
```

The following points about the field attributes should be noted:

- The *NEWPAGE attribute indicates that a new page should be started whenever the company number changes.
- The *TOTLEVELn attribute indicates the total "level breaks" that are required. In this case totals are required by company, division (within company), and department (within division within company).
- The *TOTAL attribute indicates the fields that are to be totaled. In this case the expenditure and revenue fields are to be totaled.

Refer to [Field Attributes](#) and their use for more details.

Note also that LANSAs does not sort the data. The data is printed in the same order as it is presented to the UPRINT command. It is the responsibility of the programmer to ensure that the new page and total level attributes "make sense" with regard to the order in which the information is printed.

Note this RDML program could also have been coded as:

```
SELECT  FIELDS(#COMP #DIV #DEPT #EXPEND #REVNU) FROM_FI  
UPRINT  FIELDS((#COMP *TOTLEVEL1 *NEWPAGE) (#DIV *TOTLE  
ENDSELECT  
  
ENDPRINT
```

7.98 USE

The USE command is used to invoke a [Built-In Function](#) (BIF). Arguments may be passed to the Built-In Function and values may be returned by the Built-In Function.

Also See

[7.98.1 USE Parameters](#)

[7.98.2 USE Examples](#)

Required

USE ----- BUILTIN ----- built in function name ----->

Optional

*>-- WITH_ARGS ---- list of arguments ----->
| expandable group expression |
----- 20 maximum -----*

*>-- TO_GET ----- list of field names -----|
| expandable group expression |
----- 20 maximum -----*

7.98.1 USE Parameters

BUILTIN

TO_GET

WITH_ARGS

BUILTIN

Specifies the name of the Built-In Function that is to be invoked. The name must match one of those specified in the *Built-In Functions*.

WITH_ARGS

Optionally allows a list of up to 20 arguments to be passed to the Built-In Function.

An entry in the list of arguments may be a field name, an expandable group expression, an alphanumeric literal, a numeric literal or a system variable name. The values of entries specified in an argument list are not changed by the Built-In Function (unless the same field name is repeated in the RET_VALS list).

Entries specified in an arguments list must match in type and number those specified for the Built-In Function. Refer to the *Built-In Functions* for further details.

TO_GET

Optionally allows a list of up to 20 field names to be specified as "return values" passed back by the Built-In Function.

Some Built-In Functions can return information to the program that invoked them. In such cases the return values are mapped back into fields that are nominated in the RET_VALS list.

An entry in the list of return values **can only be a field name**. The field must be defined in the LANSAs dictionary or within the function by a DEFINE command. Expandable group expressions are allowed in this parameter.

Entries specified in a return values list must match in type and number those specified for the Built-In Function. Refer to the *Built-In Functions* for more details.

7.98.2 USE Examples

Example 1: Use the built in concatenation functions to concatenate name fields #FIRST, #SECOND and #THIRD into field #NAME in a number of different ways:

```
USE BUILTIN(CONCAT) WITH_ARGS(#FIRST #SECOND) TO_GET(#NAME)
USE BUILTIN(BCONCAT) WITH_ARGS(#FIRST #SECOND #THIRD) TO_GET(#NAME)
USE BUILTIN(TCONCAT) WITH_ARGS(#SECOND " ", " #FIRST) TO_GET(#NAME)
GROUP_BY NAME(#XG_PARMS) FIELDS(#FIRST #SECOND #THIRD)
USE BUILTIN(CONCAT) WITH_ARGS(#XG_PARMS) TO_GET(#NAME)
```

Example 2: Use Built-In Function UPPERCASE to convert the contents of field #NAME to uppercase:

```
USE BUILTIN(UPPERCASE) WITH_ARGS(#NAME) TO_GET(#NAME)
```

7.99 VALUECHECK

The VALUECHECK command is used to check a field against one or more specific values.

Also See

[7.99.1 VALUECHECK Parameters](#)

[7.99.2 VALUECHECK Examples](#)

[7.4 BEGINCHECK](#)

[7.7 CALLCHECK](#)

[7.14 CONDCHECK](#)

[7.16 DATECHECK](#)

[7.35 ENDCHECK](#)

[7.48 FILECHECK](#)

[7.78 RANGECHECK](#)

Required

```
VALUECHECK --- FIELD ----- field name -----  
----->
```

```
>-- WITH_LIST ---- compare value ----->  
      |           |  
      ----- 50 max -----
```

Optional

```
>-- IN_LIST ----- *NEXT ----->  
      *ERROR  
      *ACCEPT  
  
>-- NOT_INLIST --- *ERROR -----  
>  
      *NEXT  
      *ACCEPT  
  
>-- MSGTXT ----- *NONE ----->
```

message text

>-- *MSGID* ----- *DCU0002* ----->
message identifier

>-- *MSGF* ----- *DC@M01 . *LIBL* -----
>
message file . library name

>-- *MSGDTA* ----- *substitution variables* -----|
| *expandable group expression* |
----- *20 max* -----

7.99.1 VALUECHECK Parameters

FIELD

IN_LIST

MSGDTA

MSGF

MSGID

MSGTXT

NOT_INLIST

WITH_LIST

FIELD

Specifies the name of the field to be checked.

WITH_LIST

Specifies from 1 to 50 values that are to be checked against the field. See following examples for more details.

IN_LIST

Specifies the action to be taken if the field is found to match one of the values specified in the WITH_LIST parameter.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

NOT_INLIST

Specifies the action to be taken if the field does not match any of the values in the list specified in the WITH_LIST parameter.

If *NEXT is specified the field is considered to have passed the validation check. Processing will continue with the next RDML command.

If *ERROR is specified the field is considered to have failed the validation check. Either the message text specified in MSGTXT or the message specified in MSGID and MSGF parameters will be displayed on line 22/24 of the next screen format presented to the user. In addition the field named in the FIELD parameter will be displayed in reverse image and the screen cursor will be positioned to the first field on the screen that is in error. Processing continues with the next RDML command.

If *ACCEPT is specified the field is considered to have passed the validation check **AND** no other validation checks will be performed against the field named in the FIELD parameter within this validation block. Processing continues with the next RDML command. However, if this is another validation check against the same field it will be effectively "disabled" and not performed.

MSGTXT

Allows up to 80 characters of message text to be specified. The message text specified should be enclosed in quotes. Use either the MSGTXT parameter or the MSGID / MSGF parameters but not both.

MSGID

Allows a standard message identifier to be specified as the message that should be used. Message identifiers must be 7 characters long. Use this parameter in conjunction with the MSGF parameter.

MSGF

Specifies the message file in which the message identified in the MSGID parameter will be found. This parameter is a qualified name. The message file name must be specified. If required the library in which the message file resides can also be specified. If no library name is specified, library *LIBL is assumed.

MSGDTA

Use this parameter only in conjunction with the MSGID and MSGF parameters. It specifies from 1 to 20 values that are to be used to replace "&n" substitution variables in the message specified in the MSGID parameter.

Values in this parameter may be specified as field names, an expandable group expression, alphanumeric literals or numeric literals. They should exactly match

in type, length and specification order the format of the substitution variables defined in the message.

When a field specified in this parameter has a type of signed (also called zoned) decimal, the corresponding "&n" variable in the message should have type *CHAR (character). This may cause a problem when dealing with negative values. In this case use packed decimal format instead.

When an "&n" variable in the message has type *DEC (packed decimal) the field specified in this message must be of packed decimal type.

When using alphanumeric literals in this parameter, remember that trailing blanks may be significant. For instance, if a message is defined as:

```
"&1 are out of stock ... reorder &2"
```

where &1 is defined as (*CHAR 10) and &2 as (*DEC 7 0), then the message will NOT be issued correctly if specified like this:

```
MSGDTA('BOLTS' #ORDQTY)
```

or like this

```
MSGDTA('BOLTS  ' #ORDQTY)
```

To make LANSAs aware of the trailing blanks, the parameter must be specified like this:

```
MSGDTA("BOLTS  " #ORDQTY)
```

When expandable expressions are used, the expanded field list must not exceed the maximum number of substitution variables allowed in the parameter.

7.99.2 VALUECHECK Examples

[Structuring Functions for Inline Validation](#)

[Structuring Functions to Use a Validation Subroutine](#)

[Using the VALUECHECK Command for Inline Validation](#)

[Using the VALUECHECK Command for Validation with a Subroutine](#)

Structuring Functions for Inline Validation

Typically, functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for inline validation like this:

```
BEGIN_LOOP
REQUEST << INPUT >>
BEGINCHECK
    << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK
    << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is passed back to the REQUEST command. This happens because of the default IF_ERROR(*LASTDIS) parameter on the ENDCHECK command.

Structuring Functions to Use a Validation Subroutine

Typically functions using validation commands (e.g.: CONDCHECK, DATECHECK, FILECHECK, RANGECHECK and VALUECHECK) are structured for subroutine validation like this:

```
DEFINE FIELD(#ERRORCNT) REFFLD(#STD_NUM)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')

BEGIN_LOOP
DUNTIL COND(*NOERRORS)
REQUEST << INPUT >>
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
```

```
    << PROCESS THE VALIDATED INPUT HERE >>
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE   FIELD(#ERRORCNT) TO(0)
BEGINCHECK KEEP_COUNT(#ERRORCNT)
    << USE CHECK COMMANDS TO VALIDATE INPUT HERE >>
ENDCHECK IF_ERROR(*NEXT)
ENDROUTINE
```

If a validation command inside the BEGINCHECK / ENDCHECK command block detects a validation error control is returned to the main function loop with #ERRORCNT > 0.

Using the VALUECHECK Command for Inline Validation

This example demonstrates how to use the VALUECHECK command within the main program block to check that a department code is one of a set of values.

```
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #DEPARTMENT)

BEGIN_LOOP
REQUEST  FIELDS(#EMPNO #DEPARTMENT) BROWSELIST(#EMPBROW

BEGINCHECK
VALUECHECK FIELD(#DEPARTMENT) WITH_LIST(ADM AUD FLT GAC)
ENDCHECK

ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

If the value of #DEPARTMENT is not in the list of values specified with the 'WITH_LIST' parameter the message defined in the VALUECHECK command is issued and program control returns to the last screen displayed. In this case the last screen displayed is the REQUEST screen.

Using the VALUECHECK Command for Validation with a Subroutine

This example demonstrates how to use the VALUECHECK command inside a subroutine to check a department code is in a set of values.

After the user enters the requested details the VALIDATE subroutine is called. It checks that the value of #DEPARTMENT is in the set of values specified with the 'WITH_LIST' parameter. If it is not the message defined in the VALUECHECK command is given and the DOUNTIL loop executes again. When a value for #DEPARTMENT is entered that is in the set of specified values the DOUNTIL loop ends and processing of the verified input is done.

```
DEFINE FIELD(#ERRORCNT) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEF_COND NAME(*NOERRORS) COND('#ERRORCNT = 0')
DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #DEPARTMENT)
```

```
BEGIN_LOOP
DOUNTIL COND(*NOERRORS)
REQUEST FIELDS(#EMPNO #DEPARTMENT) BROWSELIST(#EMPBROW
EXECUTE SUBROUTINE(VALIDATE)
ENDUNTIL
ADD_ENTRY TO_LIST(#EMPBROWSE)
END_LOOP
```

```
SUBROUTINE NAME(VALIDATE)
CHANGE FIELD(#ERRORCNT) TO(0)
```

```
BEGINCHECK KEEP_COUNT(#ERRORCNT)
VALUECHECK FIELD(#DEPARTMENT) WITH_LIST(ADM AUD FLT GAC)
ENDCHECK IF_ERROR(*NEXT)
```

```
ENDROUTINE
```

7.100 WHEN

The WHEN command is used within a CASE / ENDCASE block in conjunction with other WHEN commands and an optional OTHERWISE command to condition the execution of RDML commands.

The WHEN command is used to nominate expression(s) that should be evaluated as true or false (in conjunction with the CASE command) to condition the execution of the following RDML commands.

Only one WHEN command can be evaluated to true within a CASE ENDCASE structure.

Refer to the CASE, ENDCASE and OTHERWISE commands for more details and examples of these commands.

Also See

[7.100.1 WHEN Parameters](#)

[7.100.2 WHEN Examples](#)

[7.8 CASE](#)

[7.34 ENDCASE](#)

[7.73 OTHERWISE](#)

Required

```
WHEN ----- VALUE_IS ----- 'expression' -----  
-|  
  |           |  
  |           |  
  --- 20 max ----
```

7.100.1 WHEN Parameters

VALUE_IS

Specifies from 1 to 20 expressions that should be combined with the OF_FIELD parameter of the associated CASE command. See the [7.100.2 WHEN Examples](#) for more details. Refer to [Specifying Conditions and Expressions](#) for more information about specifying conditions and expressions.

7.100.2 WHEN Examples

Refer to [7.8 CASE](#) command for examples.

8. RDMLX Commands and RDMLX Features

RDMLX Commands

RDMLX commands are only used in the Visual LANSAs Editor and development environment.

RDMLX commands may be used in components and any LANSAs objects that have been enabled for Full RDMLX. (Refer to [RDML and RDMLX Partition Concepts](#) in the Administrator Guide.)

Go to the [RDMLX Commands List](#) for a summary of the RDMLX commands.

RDMLX Features

LANSAs objects that have been enabled for Full RDMLX may use the following features:

[Intrinsic Functions](#)

[8.25 Component Variables and Values](#)

[8.26 Function Libraries](#)

[8.27 Variant Handling](#)

[8.28 Enhanced Expressions](#)

Also See

[RDML Commands](#)

8.1 ASSIGN

The LANSAC CHANGE command has been the means of assigning a value to one or more variables. In the vast majority of situations this is a cumbersome way of assigning values. Full RDMLX introduces the ASSIGN command that can be specified in a program without the command name.

```
#FULLNAME := #SURNAME.Trim + ',' + #GIVENAME.Trim  
#STD_NUM += 10  
#ADDRESS1 #ADDRESS2 #ADDRESS3 := *DEFAULT
```

The most important aspect of this command is that the command and all keywords are optional.

We recommend you only use the [CHANGE](#) command when you need control over PRECISION and ROUND_UP options. Otherwise always use ASSIGN.

Also See

[8.1.1 ASSIGN Parameters](#)

[8.1.2 ASSIGN Examples](#)

Optional

```
ASSIGN ----- Variables (variable list) -----  
>  
  
  >-- Using (operator) ----->  
  
  >-- Expression (assignment expression) -----  
>
```

8.1.1 ASSIGN Parameters

The command and all keywords are optional.

Valid Operators

The following list details the operators supported by the ASSIGN command.

- := Simple value assignment
- += The value of the variable plus the value on the right
- = The value of the variable minus the value on the right
- *= The value of the variable multiplied by the value on the right
- /= The value of the variable divided by the value on the right
- <= Assign the reference on the right to the subject

Also See

[8.28 Enhanced Expressions](#)

8.1.2 ASSIGN Examples

Simple value assignments

```
#PHBN_1.Left := 10  
#PHBN_2.Left := #PHBN_1.Left + 10  
#PHBN_3.Left := #PHBN_2.Left + 10
```

String assignment

```
#FullAddress := #Address1.RightTrim + ' ' + #Address2.RightTrim + ' ' + #Ad
```

Arithmetic assignments

```
#PHBN_1.Width += 10  
#PHBN_1.Height *= 2
```

Multiple assignments

```
#ADDRESS1 #ADDRESS2 #ADDRESS3 := *DEFAULT
```

Reference assignments

```
#REF_ONE #REF_TWO #REF_THREE <= *NULL
```

```
Mthroutine Assign_example
```

```
Define_Map For(*input) Class(#prim_objt) name(#Object) Pass(*by_Referenc
```

```
Define_Com Class(#Prim_phbn) Name(#Current_button) Reference(*dynamic
```

```
#Current_button <= #Object *as #prim_phbn
```

```
Endroutine
```

8.2 ATTRIBUTE

A new RDMLX command called ATTRIBUTE enables the assignment of declarative attributes to the features of a component class.

The ATTRIBUTE command can be coded immediately after the following commands:

- BEGIN_COM
- DEFINE_COM
- DEFINE_PTY
- DEFINE_EVT
- MTHROUTINE
- DEFINE_MAP

Visual LANSA is primarily an imperative language, but like all imperative languages it does have some declarative elements. For example, the FUNCTION command enables declarative attributes to be assigned to a function or component that are later processed by the LANSA build and runtime environments.

Through its support for attributes, Visual LANSA generalizes this capability, so that programmers can invent new kinds of declarative information, attach this declarative information to various program entities, and retrieve this declarative information at run-time. Components specify this additional declarative information by defining and using attributes.

For instance, Visual LANSA might define a WinHelpAttribute attribute that can be placed on program elements such as visual member variables, enabling developers to show Window's help for their application when the F1 key is pressed.

It also means that the declarative information is stored (and consequently copied) with the RDMLX source code to which they are directly related.

Attribute classes will play a significant role in maintaining the type library information for those component classes enabled for ActiveX integration.

Whilst it is possible manually declare Attribute statements in component source it is strongly recommended that the automated ActiveX tool is used to specify ActiveX attribute details.

Attribute classes

A component class that directly or indirectly inherits from the abstract class

#PRIM_ATTR is an attribute class. The only supported attribute classes will be supplied by LANSAs.

Attribute usage

The AttributeUsage attribute of an attribute class describes how an attribute can be used.

The constructor of the AttributeUsage attribute has a positional parameter that enables an attribute class to specify the kinds of declarations on which the attribute can be used.

The constructor of an AttributeUsage attribute also includes an optional parameter that indicates whether the attribute can be specified more than once for a given declaration.

Instance Constructors

The instance constructors of an attribute class define the required and optional parameters needed to complete the definition of an attribute.

Reflection

The component classes that support access to Visual LANSAs Types, Methods, Events, Properties and their properties will provide properties that support access to the attributes assigned to the feature.

Also See

[8.2.1 ATTRIBUTE Parameters](#)

[8.2.2 ATTRIBUTE Examples](#)

```

                                     Required
ATTRIBUTE -----CLASS -----Class Name -----
->
-----
                                     Optional
>-- DESC ----- Short Description of event ----->
>-- HELP ----- Long description of the Attribute --
>
```


8.2.1 ATTRIBUTE Parameters

CLASS

DESC

HELP

CLASS

Used to define the type of attribute.

DESC

The DESC parameter can be used to define a short description for the Attribute

HELP

The DESC parameter can be used to define a long description for the Attribute of up to 250 characters.

8.2.2 ATTRIBUTE Examples

It is strongly recommended that you let LANSAs define your attributes for you when defining components as ActiveX.

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(306) Clientwidth
```

```
Define_Pty Name(Property) Get(*auto #std_text)
```

```
Define_Evt Name(Text_Changed)
Define_Map For(*input) Class(#Std_text) Name(#Text)
```

```
Mthroutine Name(Method_1)
Define_Map For(*input) Class(#Std_text) Name(#Text)
Define_Map For(*Result) Class(#Std_text) Name(#Result)
```

```
Endroutine
```

```
End_Com
```

The form above is required to be made available as an ActiveX control. From the Edit menu, select the Set ActiveX Attributes menu option. The result will be similar to the following.

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(306) Clientwidth
Attribute Class(#PRIM_ATTR.AX_TYPELIB) Guid('{0BA92712-162E-
48CB-AF41-7087EA05BD5E}') Typelibname('LANSAs_AAA_LIB')
Attribute Class(#PRIM_ATTR.AX_CLASS) Guid('{ADC02BD2-C249-
48B1-A0B1-30900707B32C}') Progid('LANSAs.AAA')
Attribute Class(#PRIM_ATTR.AX_IN_INTERFACE) Guid('{5A2034F3-
2E84-4475-A90E-B6857E62C892}') Basedispid(0)
Attribute Class(#PRIM_ATTR.AX_EVT_INTERFACE) Guid('{58DACA90-
A2D7-48F9-B459-E56DBFE9E810}') Basedispid(0)
```

```
Define_Pty Name(Property) Get(*auto #std_text)
Attribute Class(#PRIM_ATTR.AX_IN_MEMBER) DispId(0) Name('Property')
```

```
Define_Evt Name(Text_Changed)
Attribute Class(#PRIM_ATTR.AX_EVT_MEMBER) DispId(0) Name('Text_C
Define_Map For(*input) Class(#Std_text) Name(#Text)
```

```
Mthroutine Name(Method_1)
Attribute Class(#PRIM_ATTR.AX_IN_MEMBER) DispId(1) Name('Method_
Define_Map For(*input) Class(#Std_text) Name(#Text)
Define_Map For(*Result) Class(#Std_text) Name(#Result)
```

Endroutine

End_Com

Attribute statements are inserted for the component and each of the features (properties, events and methods) defined within the source.

8.3 BEGIN_COM

BEGIN_COM begins the component definition. It has an associated END_COM statement which ends the component definition.

This command is inserted automatically by the editor when you create a new component.

If you change the default values for any of the properties of the component, these are shown in the component definition. For example if you change the Visible property to False in a form you are editing, the BEGIN_COM statement will show this:

```
BEGIN_COM VISIBLE(False)
```

Normally you change the component's properties using the Component view, but you can also type them in the Source view.

Also See

[8.3.1 BEGIN_COM Parameters](#)

[8.3.2 BEGIN_COM Examples](#)

[8.8 END_COM](#)

Optional

```
BEGIN_COM --- Property Values -----  
>  
  
  >-- HELP ----- Help text ----->  
  
  >-- PUBLIC ----- name ----->  
  
  >-- PROTECT ----- name ----->  
  
  >-- PRIVATE ----- name ----->
```

>-- *DEFAULTPTY*— *name* -----
>

>-- *OPTIONS*----- **FINAL*-----
>
 **ABSTRACT*

>-- *ROLE* ----- *Help text* -----|

8.3.1 BEGIN_COM Parameters

HELP

PUBLIC

PROTECT

PRIVATE

OPTIONS

DEFAULTPTY

ROLE

HELP

Use the HELP parameter to write a description for the component. It can be 250 characters long.

PUBLIC

Use the Public property of a form or a reusable part to make public a custom-defined event, method, or property of its ancestor which has been defined as protected in the ancestor.

A form or a reusable part can define that custom-defined events, methods and properties are not accessible from outside using the Protect property of the form or reusable part. Events, methods and properties defined as protected are visible to forms and reusable parts which inherit from it, but they are not visible to owner forms or reusable parts.

An inheriting form or a reusable part can make an inherited protected event, method or property visible to all other components by defining them as Public. The list of values for Public shows two internal system values CreateInstance and DestroyInstance. They should not be used.

The easiest way to set the Public parameter is to use display the Properties in the Details Tab. When you click on the Public property, a list of the protected events, methods and properties is displayed.

PROTECT

Use the Protect property to hide custom-defined events, properties or methods from owner reusable parts or forms. An event, method or property defined as protected is visible to inheriting reusable parts and forms. These can make the protected event, property or method visible to owners by defining it as Public.

The easiest way to set the Protect parameter is to display the Properties in the

Details Tab. When you click on the Protect property, a list of the user-defined events, methods and properties is displayed.

PRIVATE

Use the Private property of a form or a reusable part to hide a custom-defined event, method or property from other forms or reusable parts. An event, method or property defined as Private is not visible to inheriting or owning reusable parts or forms.

The easiest way to set the Private parameter is to use display the Properties in the Details Tab. When you click on the Private property, a list of the user-defined events, methods and properties is displayed.

OPTIONS

Use the Options parameter to control how the component is inherited and instantiated.

- *FINAL The component class cannot be inherited further.
- *ABSTRACT The component class cannot be instantiated. You must inherit from this class and create instances of the descendent class.

DEFAULTPTY

The default property of this component.

This means that when you pass a reference to this component as `*BY_VALUE` parameter (in [8.6 DEFINE_MAP](#)), it will automatically use the default property to get the value.

This is similar to using the default property of fields: Value. Whenever I refer to, for example, `#SALARY` in the code I am actually referring to `#SALARY.VALUE` since the `.Value` property is the `SALARY` classes default property.

ROLE

The Role parameter has a number of functions.

*Extends

*Extends comes immediately prior to the ancestor class for the component. In the example below, the component inherits from `#Prim_FORM`, the Visual LANSA primitive Form class.

```
Begin_Com Role(*EXTENDS #PRIM_FORM)
```

```
End_Com
```

*Extends corresponds directly to the ancestor property as seen in the property detailer and can be entered either in the details or directly in the source.

***Implements**

*Implement is used to specify the interfaces that the component will implement. Unlike *Extends, which is limited to a single component, *Implements can have multiple arguments as shown below.

```
Begin_Com Role(*EXTENDS #PRIM_PANL *implements  
#Prim_dc.iMonitorSubject #Prim_dc.iContextualSubject)
```

```
End_Com
```

***ListFields**

*ListFields applies when a reusable part is being used as the design element of a user defined control (UDC). When ADD_ENTRY is used to create a list entry in a UDC, the fields or group of fields specified in the variables(s) following *ListFields are initialized with the values from the corresponding fields in the component that contains the ADD_ENTRY.

```
Begin_Com Role(*EXTENDS *Implements #Prim_Tile.iTileDesign  
#PRIM_PANL *ListFields #ListFields)
```

```
Group_by Name(#ListFields) Fields(#Field #Field2 #Field3)
```

```
End_Com
```

8.3.2 BEGIN_COM Examples

Begin_Com examples of components that can be visible at run time.

```
Begin_Com Role(*EXTENDS #PRIM_FORM) Height(338) Left(118) Top(163)
End_com
```

```
Begin_Com Role(*EXTENDS #PRIM_PANL)
End_Com
```

An ancestor of PRIM_OBJT can be used to create component that has no visible portion

```
Begin_Com Role(*EXTENDS #PRIM_OBJT)
End_Com
```

You can also use your own predefined component classes as an ancestor, as below, where XXXXXXXX is the name of your component.

```
Begin_Com Role(*EXTENDS #XXXXXXXXXX)
End_Com
```

8.4 DEFINE_COM

This command is inserted automatically as you drag components to be included in the component being edited.

DEFINE_COM defines instances of other components to be included in the component being edited.

In addition to the class and the name of the component, this command shows all property values for which default values have been overridden.

In Functions that are running as *HEAVYUSAGE, the state of referenced components is retained between invocations. If the state is not to be retained, use components that are *DYNAMIC.

Also See

[8.4.1 DEFINE_COM Parameters](#)

[8.4.2 DEFINE_COM Examples](#)

Required

```
DEFINE_COM ----CLASS -----Component Class -----
----->
          *VARIANT
>-- NAME ----- Component name -----
>
>-- SCOPE ----- *DEFAULT-----
>
          *LOCAL
          *INSTANCE
          *SHARED
DEFINE_COM ----CLASS -----Component Class -----
----->
>-- REFERENCE--- *DEFAULT-----
>
          *STATIC
          *DEFERRED
```

**DYNAMIC*

Optional

>-- DESC ----- Description ----->

>-- HELP ----- Help text -----|

8.4.1 DEFINE_COM Parameters

CLASS

DESC

HELP

NAME

OPTIONS

SCOPE

REFERENCE

CLASS

CLASS is the type of the component, for example command button, check box, form etc.

For fields, the class can be a simple field such as #SALARY or any of its visualizations such as #SALARY.VISUAL or #SALARY.MYPART.

*VARIANT allows the use of a variant variable. You can read or write values from a variant variable. It has no properties. At the moment a better alternative is to use a [8.25.3 Variant Variable](#).

OPTIONS

The Options parameter can have a keyword LIST_ENTRIES with value *COMPUTE or *MAX. For example:

```
OPTIONS(*LIST_ENTRIES *MAX)
```

The *List_Entries keyword specifies the number of entries a list-type component can have. List type components are:

- Grid
- ListView
- TreeView
- ComboBox
- ListBox
- Memo
- Graph
- Property Sheet

The number of entries allowed depends on whether the list contains RDMLX fields and if the list is in a component that is enabled for Full RDMLX.

*COMPUTE sets the maximum number of entries required based on the fields in the list. If the list contains only RDML fields, the number of entries is restricted to RDML levels(limited to 32767 entries). If the list contains RDMLX fields, the list is treated as a Full RDMLX list which can have a much higher platform-dependent maximum number of entries.

*MAX sets the limit of entries to the maximum allowed by the containing component. Independent of whether the fields in the list are RDMLX or not, if the list is in an RDML component, it will be limited to RDML list entries but if it is a Full RDMLX component, it will be an RDMLX list

NAME

NAME is the unique name of this instance of a component.

SCOPE

The SCOPE parameter can be used to create component instances that are shared between different instances of owner components by specifying *SHARED as the value. When *SHARED is specified in the DEFINE_COM, only one single shared instance of a member component is created regardless of how many instances of the owner component are created.

*SHARED can only be used in DEFINE_COM commands that are placed directly after the BEGIN_COM statement of the component.

When the value of the SCOPE parameter is *DEFAULT, the scope of the DEFINE_COM command is determined by its position. If it is placed immediately after the BEGIN_COM statement, the scope is interpreted as *INSTANCE. If the DEFINE_COM command is located after an event, method or property routine, the scope is interpreted as *LOCAL.

A scope of *INSTANCE causes a new instance of a member component to be created whenever a new instance of its owner component is created. This value can only be specified in DEFINE_COM commands that immediately follow the BEGIN_COM statement.

A scope of *LOCAL causes a new local instance of a component to be created every time an event, method or property routine is executed. This value can only be specified in DEFINE_COM commands that immediately follow the event, method or property routine.

All *APPLICATION variables are identified by variable name. Therefore, two different component classes can share a component instance simply by

including a DEFINE_COM for the variable name and specifying a scope of *APPLICATION.

The first reference to an *APPLICATION scoped variable that is not *DYNAMIC will cause the component instance to be created. All other accesses retrieve that instance.

When a component instance at scope *APPLICATION is retrieved, the only checking performed is to ensure that the class of the component instance can be dynamically cast to the class specified on the variable's DEFINE_COM.

*APPLICATION variables are released when the application terminates. Care must be taken to ensure that the component classes used by an instance of a component at *APPLICATION scope are fully understood. All the component DLL's required to implement these component classes will remain in memory for the lifetime of the component instance and this could correspond to the lifetime of the application.

REFERENCE

The REFERENCE parameter is used to define how the reference to the component being defined is resolved. The reference is created to the object defined by the CLASS parameter and assigned to the variable defined by the NAME parameter in the DEFINE_COM statement.

By assigning references, you can use system resources economically because you can control when a reference to an object is created and released and thus free memory locations as they are no longer required. Also, when you are using external components and applications via ActiveX you will need to set references at run time.

*DEFAULT indicates that the default reference determined by the *SCOPE parameter is used.

*STATIC indicates that when the DEFINE_COM statement is encountered during execution, a reference is created to the component defined by the CLASS parameter and the reference is assigned to the variable defined by the NAME parameter. The CLASS parameter must define a concrete class. This is the default when SCOPE is *INSTANCE (except for forms) or *LOCAL.

*DEFERRED indicates that when the DEFINE_COM statement is encountered during execution, the component reference is set to *NULL. Then when the component used in the code, a reference is created to the component defined by the CLASS parameter and the reference is assigned to the variable defined by the NAME parameter. This means that if the code that uses this component is

not executed, no reference is created. The CLASS parameter must define a concrete class. This is the default for forms which have a SCOPE of *INSTANCE.

*DYNAMIC indicates that when the DEFINE_COM statement is encountered during execution, the component reference is set to *NULL and no reference to a component is created automatically. You must use a SET_REF command to assign a reference to the component defined by the CLASS parameter and to assign the reference to the variable defined by the NAME parameter.

RDMLX Functions can use the DEFINE_COM command. For functions that are running as *HEAVYUSAGE, the state of referenced components is retained between invocations. If the state is not to be retained, use components that are *DYNAMIC.

Note:

Any property you change for this instance of the component is shown in the DEFINE_COM statement. Default values for properties are not shown.

DESC

Use the DESC parameter to write a brief description for this instance of the component. It can be 40 characters long.

The setting of the OPTIONS parameter in the DEFINE_EVT command automatically set the value of the OPTIONS parameters for the EVTROUTINE command handling the event.

HELP

Use the HELP parameter to write a longer description for this instance of the component. The help text can be viewed using the Features option of the Help menu. It can be 250 characters long.

8.4.2 DEFINE_COM Examples

The following code is inserted automatically when a push-button is included on the form and its caption is made OK. The DisplayPosition, Left and Top properties specify where it is placed. The TabPosition indicates the order in which controls are selected on a form when the Tab key is pressed.

```
DEFINE_COM CLASS(#PRIM_PHBN) NAME(#PHBN_1) CAPTION(OK) :
```

8.5 DEFINE_EVT

The DEFINE_EVT command defines a user-defined event for a form. The form issues the specified event using the SIGNAL command.

Custom specified events are used to signal events from a form instance to an owner form in multi-form applications. See also the description of the DEFINE_MAP command to see how values can be mapped to events.

Also See

[8.5.1 DEFINE_EVT Parameters](#)

[8.5.2 DEFINE_EVT Examples](#)

[8.6 DEFINE_MAP](#)

Required

```
DEFINE_EVT ----NAME -----Event Name -----  
-->
```

Optional

```
>-- DESC ----- Description of event ----->  
>-- OPTIONS --- Message and Error clearing options-  
>  
>-- ACCESS --- *PUBLIC/*PROTECT/*PRIVATE ----  
----->  
>-- HELP ----- Help text for event -----|
```

8.5.1 DEFINE_EVT Parameters

NAME

DESC

OPTIONS

ACCESS

HELP

NAME

NAME is the unique name of an event. The name can be up to 20 characters long.

DESC

Use the DESC parameter to write a brief description for the event. It can be 40 characters long.

OPTIONS

The options parameter can have two values:

*CLEARERRORS or *CLEARERRORS clears the ShowError states on member forms
*NOCLEARERRORS

*NOCLEARERRORS stops the clearing of ShowError states on member forms

*CLEARMESSAGES or *CLEARMESSAGES clears messages on the form that is handling the event
*NOCLEARMESSAGES

*NOCLEARMESSAGES stops the clearing of messages on the form that is handling the event

The setting of the OPTIONS parameter in the DEFINE_EVT command automatically set the value of the OPTIONS parameters for the EVTROUTINE command handling the event.

ACCESS

This parameter supports one of the options from the list *PUBLIC, *PROTECT and *PRIVATE.

HELP

Use the HELP parameter to write a longer description for an event. The help text can be viewed using the Features option of the Help menu. It can be 250 characters long.

8.5.2 DEFINE_EVT Examples

This command defines an event called ADDRESSCHANGED in a form:

```
define_evt name(ADDRESSCHANGED) help('This event tells that the #ADD
```

The form also contains a SIGNAL command which triggers this event when the contents of the #ADDRESS1 field on the form are changed:

```
  EVTROUTINE HANDLING(#address1.changed)
    signal event(ADDRESSCHANGED)
  ENDROUTINE
```

In this way the form where these commands have been specified can communicate to an owner form that the contents of the #ADDRESS field on it have changed. An owner form can react to this event by having an event routine for it:

```
  EVTROUTINE HANDLING(#FormB.ADDRESSCHANGED)
    execute RfrshDtls
  ENDROUTINE
```

See also [8.6.2 DEFINE_MAP with DEFINE_EVT](#).

8.6 DEFINE_MAP

The DEFINE_MAP command defines the input and output value for an event, method or property routine.

The DEFINE_MAP statement is specified after the DEFINE_EVT statement or inside a MTHROUTINE or PTYROUTINE block. Input and output values have to be defined in separate DEFINE_MAP statements.

To pass more than one value to or from an event routine, you have to specify separate DEFINE_MAP statements for all of them in the DEFINE_EVT block. The same applies to method routines that can accept multiple input parameters. A property routine can have only one DEFINE_MAP statement for input and one for output because a property is always a single value.

Also See

[8.6.1 DEFINE_MAP Parameters](#)

[8.6.3 DEFINE_MAP in MTHROUTINE](#)

[8.6.4 DEFINE_MAP in PTYROUTINE](#)

[8.6.2 DEFINE_MAP with DEFINE_EVT](#)

[8.16 MTHROUTINE](#)

[8.18 PTYROUTINE](#)

[8.5 DEFINE_EVT](#)

Required

```
DEFINE_MAP --- FOR -----*INPUT -----
->
      *OUTPUT
      *BOTH
      *RESULT

      >---CLASS ----- Component Class -----
>
      *Variant

      >-- NAME ----- Component name -----
```

>

>-- *PASS* ----- **BY_VALUE* ----->
**BY_REFERENCE*

Optional

>-- *DESC* ----- *Description* ----->

>-- *MANDATORY* --**YES* -----

>

**NULL*
Default value

>-- *HELP* ----- *Help text* -----|

8.6.1 DEFINE_MAP Parameters

FOR

CLASS

NAME

PASS

MANDATORY

DESC

HELP

FOR

Parameter will be received by the associated routine.

Allowable values ARE:

*Input

*Output

*Both

*Result

*Input

***Output**

Parameter will be returned by the associated routine

***Both**

Parameter will be received and subsequently returned by the associated routine.

***Result**

Parameter will be returned by the associated routine.

*Result is only valid for a method routine, and only one *Result is allowed per Method. To return multiple results, use *Output.

A DEFINE_MAP of *Result indicates that a method routine has a defined result parameter. This is similar in concept to Intrinsic functions, and allows a method routine to be used as part of an expression.

CLASS

CLASS defines the type of value that is passed. The value of CLASS can be a repository-defined field. The field can be a simple field such as #SALARY or any of its visualizations such as #SALARY.VISUAL or #SALARY.MYPART.

*VARIANT allows the use of a variant variable. You can read or write values from a variant variable, but it has no properties. At the moment a better alternative is to use a [8.25.3 Variant Variable](#) (#PRIM_VAR).

Note that you can pass only a single value in a single DEFINE_MAP statement.

NAME

Name uniquely identifies the mapped value. The name can be up to 20 characters long and it has to be preceded by a hash, just like a field name.

Make sure the name is unique in your system. It should not be the same name as any other field or component in your program nor should it be the same as any field or component defined in the repository.

There are few things you should note about the name:

- When you use it in a CHANGE command to set a value for a field, you must refer to its value property:

```
change #empno #curemp.value
```

- When you want to change its value, use the SET command:

```
set com(#curemp) value('A0070')
```

- You can refer to the name without the value property in subroutines and built-in functions

```
use builtin(reverse) with_args(#curemp) to_get(#revname)
```

- The same name can be used in different event, method and property routines.
- The name should not contain any prefixes reserved for LANSA such as #PRIM_, #LANSA, #SYS, #COM and #LP.

PASS

Use this parameter to specify how the supplied parameter is mapped to the variable name.

By default the parameter is supplied `*BY_VALUE`. This means that a copy of the value is passed.

If the value is passed `*BY_REFERENCE`, the routine can access the value of the field and change it.

MANDATORY

Use this parameter to specify whether the parameter being specified is mandatory (`*YES`).

Specify `*NULL` to allow users of the method to not supply a reference parameter by default when Pass is `*BY_REFERENCE`. The method code can then use the `IF_REF` command to work out whether the parameter has been supplied or not.

If the parameter is optional, you must specify here the default value for the parameter.

The following code defines a default value for the optional parameter `#this_emp`:

```
define_map *input class(#empno) name(#this_emp) mandatory('A1234')
```

The following code defines that the default value for the optional parameter `#this_emp` is blank:

```
define_map *input class(#empno) name(#this_emp) mandatory('')
```

You can also supply a value for non-mandatory output parameters. This value is used to initialize the variable at the beginning of the routine and the value is replaced if the parameter is supplied when the routine is invoked.

The following code initializes `#mNumberOfTimes` to a default of 99 and `#mValid` to false:

```
define_map *output class(#std_count) name(#mNumberOfTimes) mandatory(99)
```

```
define_map *result class(#prim_boln) name(#mValid) mandatory(false)
```

DESC

You can specify a brief description for the mapped value using the DESC parameter. It can be 40 characters long.

HELP

You can specify a longer description for the mapped value using the HELP parameter. The help text can be viewed using the Features option of the Help menu. It can be 250 characters long.

8.6.2 DEFINE_MAP with DEFINE_EVT

Unlike standard events such as Click or Initialize, user-defined events can also receive values. To specify the value to be passed to the event, add a DEFINE_MAP statement after the DEFINE_EVT statement.

Let's use a form called Form B as an example. It has a user-defined event EMPLOYEE_CREATED which is signaled when the user has created a new employee record. The DEFINE_MAP statement defines that the event will also receive the employee number of the newly added employee.

This is the DEFINE_EVENT block in Form B:

```
define_evt name(employee_created)
define_map *input class(#empno) name(#this_emp)
```

FormB also has a SIGNAL command which signals that the event has been triggered and passes the employee number of the new employee to the event routine:

```
signal event(employee_created) this_emp(#empno)
```

The owner form of Form B has an event routine for the employee_created event which receives the employee number and maps it to a variable #TheValue and then assigns this value to field #empno on the owner form.

```
EVTROUTINE HANDLING(#FormB.employee_created) this_emp(#TheValue)
  set com(#empno) value(#TheValue)
ENDROUTINE
```

8.6.3 DEFINE_MAP in MTHROUTINE

When you create a custom-defined method using the MTHROUTINE command, you can optionally specify that the method can receive input values and return output values using a DEFINE_MAP statement.

Example 1

You could define a method to fetch details for an employee on the form. The method accepts the employee number (#curemp) as input and returns a transaction number (#trnno):

```
mthroutine name(GetInfo)
  define_map for(*input) class(#empno) name(#curemp)
  define_map for(*output) class(#STD_NUM) name(#trnno)
  change #empno #curemp.value
  fetch fields(#detflds) from_file(pslmst) with_key(#empno)
  change #STD_NUM '#STD_NUM + 1'
  set com(#trnno) value(#STD_NUM)
endroutine
```

An owner form of this form can now ask it to execute this method. It passes the current value of the #empno field and receives the transaction number. The value of the transaction number is assigned to a field #TRANSA on the owner form.

```
EVTROUTINE HANDLING(#MOVETO.Click)
  invoke #frmdetail.GetInfo curemp(#empno) trnno(#transa)
ENDROUTINE
```

Example 2

In this example form the Click event of the push button invokes the method LoadForm, passing the name of a form to be displayed and returning a reference to the created form instance.

Try copying and pasting this source code to a form component and compile it. Execute the form and use it to create and display instances of other forms by specifying the name of the form to be displayed and clicking the Load button. The form name entered must be the name of a previously created form.

```
BEGIN_COM HEIGHT(123) LEFT(296) TOP(120) WIDTH(209)
DEFINE_COM CLASS(#STD_OBJ.Visual) NAME(#STD_OBJ) CAPTION('
```

```
DEFINE_COM CLASS(#PRIM_PHBN) NAME(#PHBN_1) CAPTION('Load
```

```
* form collection counter and form collection
```

```
define #FormTot Reffld(#STD_NUM) default(0)
```

```
DEFINE_COM CLASS(#PRIM_KCOL) NAME(#FORMS) COLLECTS(#PR
```

```
define #Position Reffld(#STD_NUM) default(1)
```

```
EVTROUTINE HANDLING(#PHBN_1.Click)
```

```
Change #FormTot '#FormTot + 1'
```

```
* call the LoadForm method, pass it the name of the form to be instantiated and
```

```
Invoke #COM_OWNER.LoadForm FormName(#Std_Obj) FormReference(#F
```

```
ENDROUTINE
```

```
Mthroutine LoadForm
```

```
* receive the name of the form and return a reference of the form instance whic
```

```
Define_map *input #Std_Obj #FormName
```

```
Define_map *output #prim_form #FormReference pass(*by_Reference)
```

```
*Create an instance of the named form and set reference to it
```

```
Set_Ref #FormReference (*Create_from #FormName.Value)
```

```
Set #FormReference Left(#Position) Top(#Position)
```

```
Change #Position '#Position + 10'
```

```
Invoke #FormReference.ShowForm
```

```
Endroutine
```

```
END_COM
```

8.6.4 DEFINE_MAP in PTYROUTINE

When you define a property for a component, you can optionally use the PTYROUTINE command to specify routines for setting the value of the property and for returning its value. You only use a property routine when you want to manipulate the value in some way, for example to derive the value or format it.

The property routine that sets the value contains a DEFINE_MAP statement for specifying the type of value that can be input (*INPUT) to the routine. The property routine that returns the value of the property has a DEFINE_MAP statement defining what kind of value the routine returns (*OUTPUT).

Let's use a Form B as an example. The form has a user-defined property EMP_NAME. This property can be used by an owner form of Form B for setting the value of the employee name on Form B. The property can also be used by an owner form for retrieving the current value of the employee name on Form B. In Form B the employee name is handled as two separate fields #GIVENAME and #SURNAME.

The following code in Form B defines the property and specifies that its value is set using a property routine called SET_EMP_NAME and returned by a property routine called GET_EMP_NAME.

```
define_pty name(EMP_NAME) set(SET_EMP_NAME) get(GET_EMP_NAME)
```

Define Input Parameter

FormB contains this SET_EMP_NAME routine:

```
ptyroutine name(SET_EMP_NAME)
  define_map *input class(#fullname) name(#this_emp)
  * <<logic to split #this_emp to #GIVENAME and #SURNAME>>
endroutine
```

The DEFINE_MAP statement in this routine specifies that the EMP_NAME property can take a value which is valid for a #FULLNAME field and that the value is identified by the name #this_emp inside this routine.

An owner form can now set the value of the EMP_NAME property in Form B.

```
set #FORMB EMP_NAME(#fullname)
```

or

```
set #FORMB EMP_NAME('John Smith')
```

Define Output Parameter

FormB also contains the GET_EMP_NAME routine which returns the value of the EMP_NAME property:

```
ptyroutine name(GET_EMP_NAME)
  define_map *output class(#fullname) name(#this_emp)
  * <<logic to concatenate #GIVENAME and #SURNAME to form #this_emp>
endroutine
```

The DEFINE_MAP statement in this routine specifies that the EMP_NAME property will return a valid value for #FULLNAME field. The output value is referred to with the name #this_emp. The routine then concatenates the values of #GIVENAME and #SURNAME to form the value #this_emp which will be returned as the value of the EMP_NAME property.

An owner form can now query the value of EMP_NAME for example like this:

```
IF COND(#FormB.Emp_Name *eq 'John Smith')
```

8.7 DEFINE_PTY

The DEFINE_PTY command defines a user-defined property.

Custom specified properties are typically used to pass information from a component to an owner component in multi-form applications or from a reusable part to an owner form.

Also See

[8.7.1 DEFINE_PTY Parameters](#)

[8.7.2 DEFINE_PTY Examples](#)

Required

```
DEFINE_PTY ----NAME -----Property Name -----  
--->
```

```
>-- SET ----- *NONE ----->  
-- *AUTO --- Member Component Name----
```

```
|  
-- *REFERENCE -- #Variable -----|  
-- Property Routine -----|
```

```
>-- GET ----- *NONE ----->  
-- *AUTO --- Member Component Name----
```

```
|  
-- *REFERENCE - #Variable -----|  
-- *COLLECTION - #CollectionVariable -|  
-- Property Routine -----|
```

Optional

```
>-- DESC ----- Description of property ----->
```

```
>-- ACCESS ---- *PUBLIC/*PROTECT/*PRIVATE ---
```

```
----->
```

>-- *HELP* ----- *Help text for property* -----|

8.7.1 DEFINE_PTY Parameters

NAME

SET

GET

DESC

ACCESS

HELP

NAME

NAME is the unique name of a property. It can be up to 20 characters long.

SET

SET specifies what happens when an owner component sets the value of this property.

*NONE	Indicates that an owner component cannot set the value of this property.
*AUTO and component name	Indicates that an owner component can set the value of this property. The value which is passed to this property is automatically assigned to the component specified in this parameter.
*REFERENCE #variable	Sets a reference to the component defined in the variable. The variable must be defined as *DYNAMIC.
Property Routine	Indicates that an owner component of this component can set the value using a property routine. The routine is specified using the PTYROUTINE command in the form which owns the property. This option is used only when the value returned has to be derived or formatted.

GET

GET specifies what happens when an owner component retrieves the value of

this property.

*NONE	Indicates that other components cannot query the value of this property.
*AUTO and component name	Indicates that the current value of the component specified in this parameter is returned to the component querying the information.
*REFERENCE #variable	Returns a reference to the component defined in the variable. The variable must be defined as *DYNAMIC.
*COLLECTION #CollectionVariable	Returns a read-only reference to the collection specified in #CollectionVariable. You can use these kinds of properties to access a collection to do look-ups or to loop through the collection contents. This property does not allow you to change the contents of the collection.
Property Routine	Indicates that a value will be determined by a property routine. The routine must be specified in the component using the PTYROUTINE command. This option is used only when the value returned has to be derived or formatted.

DESC

Use the DESC parameter to write a brief description for the property. It can be 40 characters long.

ACCESS

This parameter supports one of the options from the list *PUBLIC, *PROTECT and *PRIVATE.

HELP

Use the HELP parameter to write a longer description for a property. The help text can be viewed using the Features option of the Help menu. It can be 250 characters long.

8.7.2 DEFINE_PTY Examples

Example 1

This command defines a property STREETNO for Form B. When an owner form sets the value of this property, the value is assigned automatically to the #address1 field on Form B. When an owner form queries the value of this property, the current value of the #address1 field on Form B is returned.

```
define_pty name(STREETNO) set(*auto #address1) get(*auto #address1)
```

An owner form can now set the value of the property like this:

```
set COM(#FORMB) STREETNO('58 Surrey St')
```

Or if it contains an #ADDRESS1 field, it can pass the current value of the field:

```
set COM(#FORMB) STREETNO(#ADDRESS1)
```

Example 2

Define a property that gets a reference to button #PHBN:

```
Define_Pty Name(Button1) Get(*Reference #Phbn_1)
```

The Button1 property of the owner component can then be used to access #Phbn_1, for example to change its Parent property:

```
Set Com(#COM_OWNER.Button1) Parent(#GPBX_1)
```

Example 3

Get a reference to a collection:

```
Define_Pty Name(TheCollection) Get(*Collection #Collection)
```

It then iterates through the collection using the reference and adds the value and the caption of the collection items to a list:

```
For Each(#Current) In(#COM_OWNER.TheCollection) Key(#CurrentKey)
```

```
Change Field(#STD_COUNT) To('#CurrentKey.Value')
```

```
Change Field(#STD_DESCS) To('#Current.Caption')
```

Add_Entry To_List(#GRID_1)

Endfor

8.8 END_COM

This command is inserted automatically by the editor.

END_COM ends the component definition. It has a matching a BEGIN_COM statement which starts the component definition. END_COM is always the last statement in a component.

Also See

[8.8.1 END_COM Parameters](#)

[8.8.2 END_COM Examples](#)

[8.3 BEGIN_COM](#)

END_COM ----- no parameters -----

|

8.8.1 END_COM Parameters

There are no END_COM parameters.

8.8.2 END_COM Examples

Refer to the [The Role parameter has a number of functions.](#) for example of the END_COM command.

8.9 ENDFOR

ENDFOR ends a FOR loop. It has a matching a FOR statement which starts the loop.

Also See

[8.9.1 ENDFOR Parameters](#)

[8.9.2 ENDFOR Examples](#)

[8.12 FOR](#)

ENDFOR ----- no parameters -----

|

8.9.1 ENDFOR Parameters

There are no ENDFOR parameters.

8.9.2 ENDFOR Examples

Refer to the [8.12.2 FOR Examples](#) for example of the ENDFOR command.

8.10 ENDROUTINE

The ENDROUTINE command is used to end an EVTROUTINE, MTHROUTINE or PTYROUTINE block.

It is inserted automatically together with the EVTROUTINE command as you click on an event in the Event tab

Also See

[8.10.1 ENDROUTINE Parameters](#)

[8.10.2 ENDROUTINE Examples](#)

[ENDROUTINE \(in RDML Commands\)](#)

[8.16 MTHROUTINE](#)

[8.18 PTYROUTINE](#)

[8.24 WEBROUTINE](#)

ENDROUTINE ----- no parameters -----

|

8.10.1 ENDROUTINE Parameters

There are no ENDROUTINE parameters.

8.10.2 ENDROUTINE Examples

Refer to the [8.11.2 EVTROUTINE Examples](#), [8.16 MTHROUTINE](#), [8.18 PTYROUTINE](#) or [8.24 WEBROUTINE](#) for example of the ENDROUTINE command.

8.11 EVTROUTINE

This command is inserted automatically together with the ENDROUTINE command as you click on an event in the Event tab.

EVTROUTINE defines an event handling routine.

Also See

[8.11.1 EVTROUTINE Parameters](#)

[8.11.2 EVTROUTINE Examples](#)

[8.10 ENDROUTINE](#)

Required

```
EVTROUTINE ----HANDLING -----  
ComponentName.EventName ----->
```

Optional

```
>-- COM_SENDER – name -----  
>  
>-- COM_CURSOR – option name -----  
>  
>-- OPTIONS ---- message and error clearing options -  
|
```

8.11.1 EVTROUTINE Parameters

HANDLING

COM_SENDER

COM_CURSOR

OPTIONS

HANDLING

HANDLING specifies the component and the event to be handled. You can enter up to 50 combinations of components and events in this parameter.

The event name is qualified by the component name.

COM_SENDER

COM_SENDER can be used to generically refer to components that signaled an event.

The COM_SENDER parameter is used to give a generic name to all components that signal an event and then inside the event routine this generic name can be used to refer to any of the signaling components.

The components can be individual components or instances of a collection.

COM_CURSOR

COM_CURSOR can be used to control the behavior of the desktop cursor during busy operations. It takes the following options:

*DEFAULT	Same as *DELAY_01.
*NEVER	No busy cursor
*IMMEDIATE	Show busy cursor immediately
*DELAY_01	Shows busy cursor if the activity takes longer than 1 second.
*DELAY_02	Shows busy cursor if the activity takes longer than 2 seconds.
*DELAY_04	Shows busy cursor if the activity takes longer than 4 seconds.

OPTIONS

The Options parameter can have two values:

<p>*CLEARERRORS or *NOCLEARERRORS</p>	<p>*CLEARERRORS clears the ShowError states on member forms</p> <p>*NOCLEARERRORS stops the clearing of ShowError states on member forms</p>
<p>*CLEARMESSAGES or *NOCLEARMESSAGES</p>	<p>*CLEARMESSAGES clears messages on the form that is handling the event</p> <p>*NOCLEARMESSAGES stops the clearing of messages on the form that is handling the event</p>

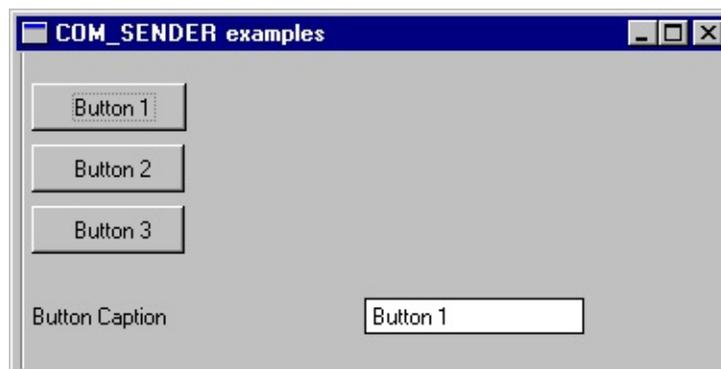
8.11.2 EVTROUTINE Examples

The following code displays a message saying "hello" when the user clicks on #HelloBtn or selects the menu item #HelloMit :

```
EVTROUTINE HANDLING(#HelloBtn.Click #HelloMit.Click)
    MESSAGE MSGTXT('hello')
ENDROUTINE
```

The following code first defines a generic name #AnyComponent for the three push-buttons #PHBN_1, #PHBN_2, #PHBN_3 which Click event is being processed. Then it sets the value of the #Button_Caption field to the caption of the button which is clicked.

```
EVTROUTINE HANDLING(#PHBN_1.Click #PHBN_2.Click #PHBN_3.Cli
    #Button_Caption := #AnyComponent.Caption
ENDROUTINE
```



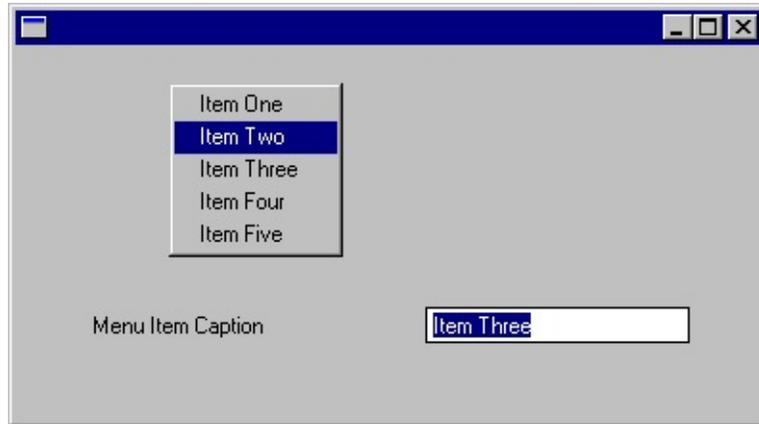
The following code defines a pop-up menu and a collection for its menu items. When the form is initialized five menu items are created in the collection. In the event routine of the Click event of the menu item collection the items are given a generic name #AnyOfTheGroup. This name is then used to assign the caption of the clicked menu item to a field named #M_Caption.

```
DEFINE_COM CLASS(#PRIM_PMNU) NAME(#PMNU_1)
DEFINE_COM CLASS(#PRIM_KCOL) NAME(#ITEMS) COLLECTS(#PRI

EVTROUTINE HANDLING(#COM_OWNER.Initialize) OPTIONS(*NOCLI
    Set Com(#Items<1>) Caption('Item One') Parent(#PMNU_1)
    Set Com(#Items<2>) Caption('Item Two') Parent(#PMNU_1)
    Set Com(#Items<3>) Caption('Item Three') Parent(#PMNU_1)
```

```
Set Com(#Items<4>) Caption('Item Four') Parent(#PMNU_1)
Set Com(#Items<5>) Caption('Item Five') Parent(#PMNU_1)
ENDROUTINE
```

```
EVTROUTINE HANDLING(#Items<>.Click) COM_SENDER(#AnyOfTheG
Change #M_Caption #AnyOfTheGroup.Caption
ENDROUTINE
```



8.12 FOR

The FOR/ENDFOR command enables the definition of a looping block of code which can be used to iterate through user-defined collections and collections provided by LANSAs.

The loop is run once per item identified in the EACH parameter.

Also See

[8.12.1 FOR Parameters](#)

[8.12.2 FOR Examples](#)

[8.9 ENDFOR](#)

Required

```
> FOR --- EACH ----- #VariableName -----
```

```
>
```

```
>-- IN ----- #Expression ----->
```

Optional

```
>-- KEY----- #KeyVariableName -----
```

```
>
```

```
>-- OPERATION----*DEFAULT -----
```

```
|
```

```
    *INSTANCE_OF ---- #ClassName
```

```
    *KIND_OF ----- #ClassName
```

```
    *DYNAMIC ----- #VariableName
```

8.12.1 FOR Parameters

EACH

IN

KEY

OPERATION

EACH

The EACH parameter names a variable that will be defined for the scope of the FOR/ENDFOR block to be a reference to the current component being supplied by the iterator.

By default, the type of this variable is that of the type of the component being collected by the collection from which the iterator was created. This can be changed by the OPERATION(...) parameter.

IN

The IN parameter identifies the collection to be iterated through. The collections can be user-defined or primitive LANSAs collections.

User-defined collection types are:

Keyed collection	Keyed collection. Keyed collections are an unordered sequence of components identified by a key value. No duplicates of the key value are allowed. Note that because there is no predefined ordering of items in a keyed collection, there is no guaranteed order in which the items will be returned when iterating through the collection.
List collection	A List collection. List collections provide an ordered collection of components. The features of the list component are positional in nature, in reference to a given index or to the beginning or end of the list. Indexing is always relative to 1.
Array collection	An array collection. Array collections are a dynamically sized, ordered collection of components that can be located by indexing. Indexing is always relative to 1.
Sorted array collection	A sorted array collection. Sorted array collections are a dynamically sized, sorted collection of components that can be located by indexing. Indexing is always relative to 1.

Set collection	A set collection. Set collections are an unordered collection of components that cannot contain duplicates.
Dictionary collection	A dictionary collection. Dictionary collections are an unordered sequence of key-value component pairs with no duplicates of the key.
Sorted dictionary collection	A sorted dictionary collection. Sorted dictionary collections are a collection of key-value component pairs. The collection is sorted by the key component and no duplicates of the key are allowed.

Primitive LANSAs are accessed using these properties:

ComponentMembers property	The ComponentMembers property of a component provides access to the collection of all its member components. All components that are owners of other components have this property.
ComponentControls property	All composite visual components (like forms, panels, tabbed folders, etc) support the ComponentControls property which enables access to its children controls.
Items property	The Items property provides access to the items in grids, list views, tree views, tree view items, list boxes, combo boxes, property sheets and menus. For a menu the Items property provides a collection of all the menu items contained in MenuBar, Popup and SubMenu components.
Columns property	The Columns property provides access to the attributes of columns in grids, list views, tree views, list boxes, combo boxes and property sheets.
ComponentForms property	The ComponentForms property of component #SYS_APPLN provides a collection of all the forms currently realized by the application.

KEY

Some collections provide a key.

Keyed collections have a key which is the type of the field defined in the KeyedBy parameter. Other collections have keys that are simply indexes.

If you want access the current key of the current component, you specify a name for the KEY parameter and Visual LANSA will automatically provide access to the current key each time you reference the variable name in the FOR/ENDFOR block.

OPERATION

This parameter lets you do casting operations in order to select specific kinds of objects from the collection.

*DEFAULT	All items are selected.
*INSTANCE_OF ClassName	Checks if the variable is of the type identified by the class name or of the type of ancestors identified by the class name.
*KIND_OF ClassName	Checks if the variable is of the class identified by the class name value.
*DYNAMIC #VariableName	Assigns the reference contained in the variable name to each of the variables identified by the EACH parameter. No compile time checking is performed to see if the variables are compatible. At run-time, if the variable name cannot be cast to any of the variables identified by the EACH parameter, an error is raised.

8.12.2 FOR Examples

Columns Collection

This example loops through the columns of a list view and retrieves the DisplayPosition and Width attributes of the columns to a grid:

```
For Each(#Current) In(#ltvw_1.Columns)
Set Com(#STD_NUM) Value(#current.displayposition)
Set Com(#std_amnt) Value(#current.width)
Add_Entry To_List(#GRID_1)
Endfor
```

ComponentMembers Collection

This example records all the member components on a form and lists them in a grid:

```
For Each(#Current) In(#COM_OWNER.ComponentMembers)
Change Field(#STD_NAME) To('#CURRENT.NAME')
Add_Entry To_List(#GRID_1)
Endfor
```

ComponentForms Collection

This example loops through the collection of all forms currently realized by an application and records them in a grid:

```
For Each(#Current) In(#SYS_APPLN.ComponentForms)
Change Field(#STD_NAME) To('#CURRENT.ComponentTypeName')
Add_Entry To_List(#GRID_1)
Endfor
```

Items Collection

In this example clicking #PHBN_1 disables all the menu items in submenu #SMNU_1 and clicking #PHBN_2 enables the menu items:

```
Evtroutine Handling(#PHBN_1.Click)
For Each(#Current) In(#smnu_1.Items)
Set Com(#Current) Enabled(False)
Endfor
Endroutine
```

```
Evtroutine Handling(#PHBN_2.Click)
For Each(#Current) In(#smnu_1.Items)
Set Com(#Current) Enabled(True)
Endfor
Endroutine
```

8.13 IF_REF

The IF_REF command is used to compare references of component variables.

Also See

[8.13.1 IF_REF Parameters](#)

[8.13.2 IF_REF Examples](#)

[8.21 SET_REF](#)

Required

```
IF_REF --- COM ----- Variable name|variable name -----  
->  
  
>-- IS ----- Reference expression ----->  
  
>-- IS_NOT ----- Reference expression -----|
```

8.13.1 IF_REF Parameters

COM

IS and IS_NOT

COM

The COM parameter identifies one or more component variables that are the target of the comparisons.

IS and IS_NOT

Specifies the reference expression that must be true to make the IF condition true. It can be one of these:

- *NULL Checks if the variables identified by the COM parameter are set to *NULL
- *EQUAL_TO Checks if the variable identified by the COM parameter is equal to this variable. In other words the condition checks if both variables refer to the same component instance.
#VariableName
- *INSTANCE_OF Checks if the variable identified by the COM parameter is of the type identified by the class name or of the type of ancestors identified by the class name.
ClassName
- *KIND_OF Checks if the variable identified by the COM parameter is of the class identified by the class name value.
ClassName

Refer also to [Specifying Conditions and Expressions](#).

8.13.2 IF_REF Examples

This example shows a message box if a reference to component #WordApp exists:

```
If_ref Com(#WordApp) Is_Not(*null)
  Use Builtin(MESSAGE_BOX_SHOW) With_Args(OK OK Information 'Re:
Endif
```

8.14 IMPORT

The IMPORT command is used to include function libraries into an object. Function Libraries are LANSAs defined primitives that contain a set of routines and functions.

The IMPORT command is specified immediately after the FUNCTION statement and before the BEGIN_COM.

Once a Function library has been imported, the routines defined in the library can be used in expressions.

FUNCTION Options(*DIRECT)

* import the variant function library named #PRIM_LIBV

IMPORT Libraries(#PRIM_LIBV)

Also See

[8.14.1 IMPORT Parameters](#)

[8.14.2 IMPORT Examples](#)

Required

*IMPORT --- Libraries---- list of library name -----
>*

8.14.1 IMPORT Parameters

LIBRARIES

Use this parameter to specify the libraries to be imported.

There are 5 defined libraries available:

PRIM_LIBD – Date time library

PRIM_LIBI – Intrinsic library

PRIM_LIBN – Number function library

PRIM_LIBS – String function library

PRIM_LIBV – Variant function library

Note: With the exception of the Variant function library, all of the features of the libraries are available through the use of intrinsic functions. This is the recommended technique.

Variants by their nature cannot support a large array of intrinsic functions. It is necessary therefore to IMPORT the available functions such that a variant can be easily manipulated.

8.14.2 IMPORT Examples

The IMPORT command is specified immediately after the FUNCTION statement and before the BEGIN_COM. The following command imports the variants function library:

```
FUNCTION Options(*DIRECT)
* import variant library
IMPORT Libraries(#PRIM_LIBV)
```

Example of how to use the IMPORTED function. The following method receives variant. If the variant is a string, it is returned as a string in the #Result

Define_map

```
Mthroutine Name(Get_Variant_value)
Define_Map For(*Input) Class(*Variant) Name(#iVariant)
Define_Map For(*Result) Class(#Prim_alph) Name(#Result)
```

```
* Call the Varisstring function
If (VarisString( #iVariant ))
```

```
#result := VarasString( #iVariant )
```

```
Endif
```

```
Endroutine
```

8.15 INVOKE

The INVOKE command invokes a method. A method can be a standard method such as a form's Show and Hide methods, or it can be a user-defined method.

Custom-defined methods can also accept input values and can return output values. When you invoke a method like this you need to pass a value and assign a field for the returned value.

Methods are often used by an owner form to instruct a member form to perform some action. Note that a component can also invoke its own methods, for example a form can close itself by invoking its Hide method.

The specification of the INVOKE command is as follows:

```
INVOKE Method(method_expression) ParmOne(#ArgOne) ParmTwo(#ArgTwo)
```

Using Full RDMLX, without its name, the command can be entered in the following way:

```
method_expression ParmOne(#ArgOne) ParmTwo(#ArgTwo)
```

Also See

[8.15.1 INVOKE Parameters](#)

[8.15.2 INVOKE Examples](#)

Required

```
INVOKE -----Component.Method -----  
->
```

Optional

```
>---- Parameter and Value -----|
```

8.15.1 INVOKE Parameters

COMPONENT.METHOD

Parameter and Value

COMPONENT.METHOD

The method to be invoked qualified by the component name.

Parameter and Value

If an output parameter has been specified in the MTHROUTINE command, the name of the parameter and the value to be passed.

8.15.2 INVOKE Examples

This code shows MyForm using its Show method:

```
INVOKE #MyForm.ShowForm
```

This code executes a GetInfo method, passes the current value of the #employee field to it, and receives a transaction number (trrno) which it assigns to the field #transa.

```
INVOKE #frmDetails.GetInfo CurEmp(#EMPNO) trrno(#TRANSA)
```

Simple invocation using Full RDMLX:

```
#Com_Owner.StringMethod String1(#Address1) String2(#Address2) String3(#  
#Com_Owner.StringMethod String2(#Address2) String3(#Address3) String1(#
```

For more information about input parameters for methods, refer to [8.6 DEFINE_MAP](#) and [8.16 MTHROUTINE](#) in this chapter.

8.16 MTHROUTINE

The MTHROUTINE command is used to define a user-defined method. A method instructs the component to do something.

The method routine can optionally receive input parameters. For more information refer to the description of [8.6 DEFINE_MAP](#).

The MTHROUTINE must have a matching ENDROUTINE command. A method is executed using the INVOKE command.

Also See

[8.16.1 MTHROUTINE Parameters](#)

[8.16.2 MTHROUTINE Examples](#)

[8.10 ENDROUTINE](#)

[8.6 DEFINE_MAP](#)

Required

```
MTHROUTINE ---- NAME -----Routine Name -----  
----->
```

Optional

```
>-- DESC ----- Description ----->  
  
>-- OPTIONS ---- *REDEFINE/*FINAL/*ABSTRACT-  
----->  
  
>-- ACCESS ---  
- *PUBLIC/*PROTECT/*PRIVATE/*DEFAULT--->  
  
>-- HELP ----- Help text-----|
```

8.16.1 MTHROUTINE Parameters

NAME

DESC

OPTIONS

ACCESS

HELP

NAME

The name of the method routine. This name can be up to 20 characters long.

DESC

Use the DESC parameter to write a brief description for the method. It can be 40 characters long.

OPTIONS

The value of the OPTIONS parameter determines whether a method inherited from an ancestor can be redefined:

- *REDEFINE The method is redefining (overriding) the implementation of a method in a component's ancestor.
- *FINAL The method cannot be redefined by components that inherited from the component defining the method.

ACCESS

This parameter supports one of the options from the list *PUBLIC, *PROTECT and *PRIVATE.

HELP

Use the Help parameter to write a longer description for a method. The help text can be viewed using the Features option of the Help menu. It can be 250 characters long.

8.16.2 MTHROUTINE Examples

You can define a user-defined method that instructs a form to fetch details from a file:

```
MTHROUTINE Name(GetInfo) Help('This method gets the details of the emp
  FETCH Fields(#detflds) From_File(pslmst) With_Key(#empno)
ENDROUTINE
```

An owner form can now instruct the form to perform this action by invoking the method:

```
INVOKE #frmDetail.GetInfo
```

For an example of how to pass values to a method routine, see [8.6.3 DEFINE_MAP in MTHROUTINE](#).

8.17 PERFORM

The PERFORM command enables the calling of a component method, library function or intrinsic feature where the routine being called does not require any parameters or the parameters are supplied as a parameter list enclosed in parentheses.

The specification of the PERFORM command is as follows:

PERFORM EXPRESSION(expression)

The most important aspect of this command is that the command and all keywords are optional.

Also See

[8.17.1 PERFORM Parameters](#)

[8.17.2 PERFORM Examples](#)

Required

PERFORM --- Expression---- Expression ----->

8.17.1 PERFORM Parameters

The command and all keywords are optional.

8.17.2 PERFORM Examples

Simple perform operations:

```
#Com_Owner.Realize
```

```
#Com_Owner.SetFocus
```

```
#Com_Owner.StringMethod(#Address1 #Address2 #Address3)
```

8.18 PTYROUTINE

The PTYROUTINE command is used to create a routine to manipulate the value of a user-defined property.

When you define a property for a form using the DEFINE_PTY command, you can use the PTYROUTINE command to specify routines for setting the value of the property and for returning its value. You do this when you want to manipulate the value of the property. For instance you can:

- Split the value that has been passed to the property and assign the resulting values to two or more fields. For example you could do this by splitting the value of an Address property into separate fields for street name and number, city, state and country and postcode.
- Concatenate the values of several fields to form the value of a property when it is retrieved.
- Calculate or otherwise derive the value of the property.

The input and output values for a property routine are defined using the DEFINE_MAP command. For more information refer to the description of [8.6 DEFINE_MAP](#) in this chapter.

It is worth noting that the custom-defined properties (DEFINE_PTY) and the property routines are all contained inside the form. As a result the form can receive property values from its owner forms as well as return them, but the way this value is handled inside the form is not visible to the owner forms. As a consequence, you can change the way the form handles the property value without having to make any change to forms that use this property.

For instance a form which has an Address property could be changed so that post code and state information which was previously handled as one field would now be handled as two fields. As a result the property routine in the form would have to be changed, but no change would be required in other forms that set or retrieve the value of the Address property because the property itself does not change.

The name of a PTYROUTINE can be the value of the SET or GET parameters in a DEFINE_PTY command. See the description of [8.7 DEFINE_PTY](#).

Also See

[8.18.1 PTYROUTINE Parameters](#)

[8.6.4 DEFINE_MAP in PTYROUTINE](#)

8.10 ENDROUTINE

Required

PTYROUTINE ---- NAME -----Routine Name -----

|

8.18.1 PTYROUTINE Parameters

NAME

NAME

The name of the routine.

8.18.2 PTYROUTINE Examples

See [8.6.4 DEFINE_MAP](#) in PTYROUTINE.

8.19 SELECT_SQL Free Format

There are two forms of the SELECT_SQL command. This section describes the free format version which allows any SQL that is valid for the particular database engine. No parsing is performed of the SQL either at compile time or runtime. The entered SQL command is passed exactly as it is to the database engine. It is the responsibility of the RDML programmer to ensure that the data returned by the database engine matches the list of fields in the FIELDS parameter. See [SELECT_SQL](#) for the other form of SELECT_SQL.

This form of the SELECT_SQL command can only be used in RDMLX functions and components.

The SELECT_SQL command is used in conjunction with the ENDSELECT command to form a "loop" to process one or more rows (records) from one or more tables (files).

For example, the following SELECT_SQL / ENDSELECT loop selects all values of product and quantity from the table ORDLIN and places them, one by one, in a list:

```
----> DEF_LIST NAME(#ALIST) FIELDS(#PRODUCT #QUANTITY)
--> SELECT_SQL FIELDS(#PRODUCT #QUANTITY)
|           USING("SELECT "PRODUCT", "QUANTITY" FROM "MYDTAL
|
|           ADD_ENTRY(#ALIST)
|
---- ENDSELECT
```

Before attempting to use free format SELECT_SQL you must be aware of the following:

1. Information accessed via SELECT_SQL is for read only. If you use INSERT or UPDATE statements in your USING parameter you do so at your own risk.
2. SELECT_SQL does not use the IO Modules/OAMs so it bypasses the repository validation and triggers.
3. The SELECT_SQL command is primarily intended for performing complex extract/join/summary extractions from one or more SQL database tables (files) for output to reports, screens or other tables. It is not intended for use in high volume or heavy use interactive applications.

With that intention in mind, it must be balanced by the fact that

SELECT_SQL is a very powerful and useful command that can vastly simplify and speed up most join/extract/summary applications, no matter whether the results are to be directed to a screen, a printer, or into another file (table).

4. The SELECT_SQL command provides very powerful database extract/join/summarize capabilities that are directly supported by the SQL database facilities. However, the current IBM i implementation of SQL may require and use significant resource in some situations. It is entirely the responsibility of the user to compare the large benefits of this command, with its resource utilization, and to decide whether it is being correctly used. One of the factors to consider is whether the USING parameter uses any non-key fields. If it does, then SELECT_SQL will probably be quicker than SELECT. Otherwise SELECT will probably be quicker. This is especially important when developing the program on Visual LANSA first with the intention of also running it on IBM i. This is because Visual LANSA has much fewer performance differences between SELECT and SELECT_SQL.
5. This section assumes that the user is familiar with the SQL 'SELECT' command. This section is about how the SQL 'SELECT' command is accessed directly from RDML functions, not about the syntax, format and uses of the SQL 'SELECT' command.

If your command is incorrect then the following diagnosis is possible:

- A useful technique when working with SQL is to use interactive SQL to "test case" your command (and its syntax) before compiling it into a SELECT_SQL command.
- At execution time. Compiling the SELECT_SQL free format command proves very little. Almost all the parsing is performed by the SQL database engine. In this case examine all the resulting error messages for the exact cause.

When dealing with an execution time error, the use of trace on the function will allow the capture of the exact SQL that the SELECT_SQL command has generated. Open the latest trace file and search for "***ERROR". This will be the same text as in the error messages. Go back 8 lines or so to the "Preparing" message and you will find the SELECT statement that caused the error. You can copy and paste this into interactive SQL to further diagnose the problem.

- One of the most common execution errors, apart from a syntax errors is that

the list of fields does not match the data returned by the SQL statement in the USING parameter.

- When reporting issues with SELECT_SQL to support you must provide the trace file and the generated C source code.

The extensive use of the SELECT_SQL command is **not recommended** for the following reasons:

- The SQL access commands are imbedded directly into the RDML function. DBMS access is direct and not done via IOM/OAM access routines. This approach may compromise the use of before and after read triggers and the use of the "thin client" designs implemented via LANSA/SuperServer.
- If the contents of SELECT_SQL is sourced from a field on a screen then it is possible for an end user to perform more than a select. It is especially easy in this Free Format version where this code is possible:

```
REQUEST FIELD(#ANYSQL)
Select_Sql Fields(#STD_NUM) Using(#ANYSQL)
endselect.
```

and the end user could enter this on the screen: "delete from mylib.afile;select count(*) from mylib.afile"

- The use of imbedded SQL features and facilities may introduce platform dependencies into your applications. Not all SQL facilities are supported by all DBMSs. By bypassing the IOM/OAM associated with the table, you are bypassing the feature isolation it provides. Using SQL features and facilities that are DBMS defined, platform dependent extensions, is solely at the discretion of, and the responsibility of, the application designer.
- Where SELECT_SQL is to be used, you should isolate the use within a specific function, separate from any user interface operations. This will allow the function to be invoked as an "RPC" (Remote Procedure Call) in the client design models.

Portability Considerations Do NOT use this command to connect from Visual LANSA to a database on the IBM i. If you use the SELECT_SQL command to connect from Visual LANSA to an IBM i Database, it will access the Database on the PC and not on the IBM i. For this type of connection, you should use a remote procedure call (i.e call_server_function).

Also See

[8.19.1 SELECT_SQL Free Format Parameters](#)

8.19.2 SELECT_SQL Free Format Examples

8.19.3 SELECT_SQL Free Format References

8.19.4 SELECT_SQL Free Format Coercions

Required

SELECT_SQL --- FIELDS ----- field name -----

>-- USING ----- SQL select command -----
>

Optional

>-- FROM_FILES --- file name ----->
 | |
 ----- 20 max-----

>-- IO_STATUS ---- field name ----->
 *STATUS

*>-- IO_ERROR ----- *ABORT -----|*
 *NEXT
 *RETURN
 label

8.19.1 SELECT_SQL Free Format Parameters

FIELDS

FROM_FILES

IO_ERROR

IO_STATUS

USING

FIELDS

Specifies the fields that will receive the result of the SQL command specified in the USING parameter. This parameter can only contain fields, groups and expandable groups

All columns nominated by this parameter must be defined in the current function or in the LANSAs data dictionary as valid RDML variables.

Fields of type BLOB and CLOB are not supported in the SELECT_SQL command. If one is specified a fatal error will occur when the command is compiled.

USING

The SQL_SELECT USING parameter can be any valid [enhanced expression](#) . It is best to use single quotes to delimit strings so that double quotes can be used around the SQL identifiers. This means that single quotes around string literals must be doubled up. The first and second examples have the same result except that the last single quote to terminate the SQL literal is specified in two different ways - either "" or """". The fourth example puts the value and the quotes into a work field which may be the easiest method to read and maintain.

```
... USING('SELECT "EMPNO" FROM "XDEMOLIB"."PSLMST" WHERE "E
```

```
... USING('SELECT "EMPNO" FROM "XDEMOLIB"."PSLMST" WHERE "E
```

```
... USING('SELECT "EMPNO" FROM "XDEMOLIB"."PSLMST" WHERE N
```

```
... #STD_TEXT := ""%a""
```

```
... USING('SELECT "EMPNO" FROM "XDEMOLIB"."PSLMST" WHERE N  
+ #STD_TEXT)
```

The SQL language uses double quotes (the quote character may differ on some databases) to surround identifiers that might otherwise be interpreted as SQL syntax. By quoting identifiers you are assured that the identifiers will not clash with any SQL syntax on any database.

The USING parameter does not support embedded fields (e.g. :KARTIC) like the WHERE parameter does.

Portability

Considerations Visual LANSAs provides access to multiple databases using Visual LANSAs Other Files. Visual LANSAs Other files can be used in SELECT_SQL, but they must all be from the same database. If a Visual LANSAs Other File is in the same database as a LANSAs file, then the two files can be used in the same SELECT_SQL command

SQL Table names may differ from the LANSAs file name, for example when an @, # or \$ is in the name. This name may also be different between different operating systems. If the SQL Command is intended to be executed on multiple platforms ensure that the table names are either the same or they are specified as a variable in the USING parameter

USING clause hints

It is usually necessary to specify the collection that the table is in. This is not necessary in the WHERE parameter because LANSAs parses the SQL and determines the correct collection to use.

All the identifiers must be spelt exactly as required by the database. For example the LANSAs name for an Other File may be different to the actual table name if the table name already exists or its longer than 10 characters. It's the table name that must be specified. The following example uses a table named "Long Table Name With Spaces" with columns named "Long Field Name 1" and "Long Field Name 2". LANSAs has loaded the table as "LONG_TABLE" and named the fields "LONG_FIEL" and "LONG_FIE1".

```
SELECT_SQL FIELDS(#LONG_FIEL #LONG_FIE1)
FROM_FILES((LONG_TABLE)) IO_ERROR(*NEXT) USING('select "Long
```

Field Name 1", "Long Field Name 2" FROM LX_DTA."Long Table Name With Spaces")

When searching for data using the like condition, characters with special meaning to SQL need to be escaped if they need to be taken literally. For example, the character '_' matches any character. To literally match '_' then the following syntax needs to be used. This will find all states that start with 'B_':

```
CHANGE
FIELDS(#TABLE) TO('SELECT "CUSTNUM" FROM "XDEMOLIB"."CUSTC
')
CHANGE FIELDS(#SELECTION) TO('WHERE STATE LIKE "B!_%" ESCAP
SELECT_SQL FIELDS(#CUSTNUM") USING(#TABLE + #SELECTION)
    DISPLAY FIELDS(#CUSTNUM)
ENDSELECT
```

Note: This nominates the exclamation mark as the escape character. Any "normal" character not greater than 127 in the ASCII table can be used. (Characters %,_,[do not work on all DBMS systems and so are not recommended.)

This has been tested on ASA, DB2400, SQL Server, and Oracle. The only exception is MS Access, where instead you need to use [] around the character to be escaped. For example: WHERE "STATE" LIKE 'B[_]%'

For further details about specifying conditions, refer to [Specifying Conditions and Expressions](#). For further information about the structure of this clause, refer to the SQL guides.

RDMLX IBM i Other Files with Unicode Fields

SQL on IBM i cannot compare a graphic unicode field directly to a string literal or a character column; a conversion error occurs.

There are three ways of converting the expression to Unicode to avoid the

conversion error:

1. Pass the literal as a Unicode (UX'ssss') literal. For example, instead of:

```
'WHERE "UNIFLD" LIKE "C%'
```

try

```
'WHERE "UNIFLD" LIKE UX"00430025").
```

2. Convert the literal or column using SQL functions so it becomes a Unicode expression. For example, CHARFLD is a Character column. Instead of

```
'WHERE "UNIFLD" = "CHARFLD")
```

try

```
'WHERE "UNIFLD" = CAST "CHARFLD" AS GRAPHIC(6) CCSID  
13488)')
```

For further details, refer to the IBM manual *DB2 UDB for IBM SQL Reference*.

FROM_FILES

Refer to [Specifying File Names](#).

This is an optional parameter, unlike the parameter of the same name in the form of the SELECT_SQL command which is parsed by LANSAs. This is because the file name used here does not effect the file accessed at runtime. It has two purposes: the first is for use in accessing Other Files that are in other databases. By specifying the file name LANSAs will be able to locate the database connection information that was used to load the Other File into LANSAs. This can be further refined by using the DEFINE_DB_SERVER and CONNECT_FILE BIFs.

The second purpose is to generate cross reference information so that use of the files in the USING parameter can be traced. This can later be used for impact analysis though clearly it is a manual cross reference and so relies on the programmer to keep it up to date. LANSAs suggests that it should be made a program documentation rule.

Examples

```
... FROM_FILES(ORDLIN)  
... USING('SELECT * FROM "MYLIB"."ORDLIN", "MYLIB"."ORDDTL"  
..... WHERE "MYLIB"."ORDLIN"."CUSTNO" = "MYLIB"."ORDDTL"."C
```

IO_STATUS

Specifies the name of a field that is to receive the "return code" that results from

the I/O operation.

If the default value of *STATUS is used the return code is placed into a special field called #IO\$STS which can be referenced in the RDML program just like any other field.

If a user field is nominated to receive the I/O return code it must be alphanumeric with a length of 2. Even if a user field is nominated the special field #IO\$STS is still updated.

Refer to [I/O Command Return Codes Table](#) for values.

IO_ERROR

Specifies what action is to be taken if an I/O error occurs when the command is executed.

An I/O error is considered to be a "fatal" error. Some examples are file not found, file is damaged, file cannot be allocated. These types of errors stop the function from performing any processing at all with the file involved.

If the default value of *ABORT is used the function will abort with error message(s) that indicate the nature of the I/O error.

*NEXT indicates that control should be passed to the next command.

*RETURN specifies that in a program mainline control is to be returned to the caller and in a subroutine control is to be returned to the caller routine or the program mainline.

If none of the previous values are used you must nominate a valid command label to which control should be passed.

8.19.2 SELECT_SQL Free Format Examples

[Using SELECT_SQL With the DISTINCT Option](#)

[Using SELECT_SQL With Calculations](#)

[Using SELECT_SQL With AND and OR Operators](#)

[Using SELECT_SQL With the BETWEEN Operator](#)

Using SELECT_SQL With the DISTINCT Option

This example demonstrates how to use the SELECT_SQL command with the DISTINCT option to eliminate duplicate field values. The use of the standard SELECT_SQL command without any extra options is also demonstrated.

```
DEF_LIST  NAME(#EMPBROWSE) FIELDS(#NDSTEMPNO #DSTEMPN
DEFINE    FIELD(#HEADING1) TYPE(*CHAR) LENGTH(79) INPUT_ATI
DEFINE    FIELD(#NDSTEMPNO) REFFLD(#EMPNO) COLHDG('Employ
DEFINE    FIELD(#DSTEMPNO) REFFLD(#EMPNO) COLHDG('Employee
DEFINE    FIELD(#ENTRYNO) TYPE(*DEC) LENGTH(5) DECIMALS(0)
```

```
CHANGE    FIELD(#HEADING1) TO('"This function uses SELECT_SQL fr
```

```
BEGIN_LOOP
EXECUTE   SUBROUTINE(NOTDISTINC)
EXECUTE   SUBROUTINE(DISTINCT)
DISPLAY  FIELDS(#HEADING1) DESIGN(*DOWN) IDENTIFY(*NOID)
END_LOOP
```

```
SUBROUTINE NAME(NOTDISTINC)
CLR_LIST  NAMED(#EMPBROWSE)
CHANGE    FIELD(#DSTEMPNO) TO(*NULL)
SELECT_SQL FIELDS(#EMPNO) USING('SELECT "EMPNO" FROM "XD
CHANGE    FIELD(#NDSTEMPNO) TO(#EMPNO)
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
ENDROUTINE
```

```
SUBROUTINE NAME(DISTINCT)
CHANGE    FIELD(#ENTRYNO) TO(1)
SELECT_SQL FIELDS(#EMPNO) USING('SELECT DISTINCT "EMPNO"
GET_ENTRY NUMBER(#ENTRYNO) FROM_LIST(#EMPBROWSE)
```

```

CHANGE  FIELD(#DSTEMPNO) TO(#EMPNO)
UPD_ENTRY IN_LIST(#EMPBROWSE)
CHANGE  FIELD(#ENTRYNO) TO('#ENTRYNO + 1')
ENDSELECT
ENDROUTINE

```

Using SELECT_SQL With Calculations

This example demonstrates how calculations can be used on data retrieved by the SELECT_SQL command.

```

DEF_LIST  NAME(#EMPBROWSE) FIELDS(#SURNAME #SALARY #STD_AMNT)
DEFINE    FIELD(#HEADING1) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(1)
DEFINE    FIELD(#HEADING2) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(2)
DEFINE    FIELD(#HEADING3) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(3)

OVERRIDE  FIELD(#STD_AMNT) COLHDG('Salary + 10%')

CHANGE    FIELD(#HEADING1) TO('"This function uses SELECT_SQL fr
CHANGE    FIELD(#HEADING2) TO('"This shows a list of employee surnar
CHANGE    FIELD(#HEADING3) TO('"This can be done with one SELECT_

BEGIN_LOOP
CLR_LIST  NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#SURNAME #SALARY #STD_AMNT)
           USING('SELECT "SURNAME", "SALARY", "SALARY" * 1.10 FRO
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
DISPLAY  FIELDS(#HEADING1 #HEADING2 #HEADING3) DESIGN(*I
END_LOOP

```

Using SELECT_SQL With AND and OR Operators

This example demonstrates how the SLECT_SQL command can be used with AND and OR operators to conduct more complex queries.

```

DEF_LIST  NAME(#EMPBROWSE) FIELDS(#EMPNO #ADDRESS2 #SALARY)
DEFINE    FIELD(#HEADING1) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(1)
DEFINE    FIELD(#HEADING2) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(2)
DEFINE    FIELD(#HEADING3) TYPE(*CHAR) LENGTH(79) INPUT_ATTRIBUTES(3)

```

```

CHANGE FIELD(#HEADING1) TO("This function uses SELECT_SQL fr
CHANGE FIELD(#HEADING2) TO("This lists all employees who either h
CHANGE FIELD(#HEADING3) TO("or who live in SEVEN HILLS. This

```

```

BEGIN_LOOP
CLR_LIST NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#EMPNO #SURNAME #ADDRESS2 #SALARY)
        USING('SELECT "EMPNO", "SURNAME", "ADDRESS2", "SALARY"
                WHERE (("SALARY" > 10000) AND ("SALARY" < 20000))
                OR ("ADDRESS2" = "SEVEN HILLS.")')
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
DISPLAY FIELDS(#HEADING1 #HEADING2 #HEADING3) DESIGN(*I
END_LOOP

```

Using SELECT_SQL With the BETWEEN Operator

This example demonstrates the use of the SELECT_SQL command with the BETWEEN operator. The BETWEEN operator can be used in the WHERE clause to retrieve data between specified values. It can also be used to retrieve data excluding that between specified values.

```

DEF_LIST NAME(#EMPBROWSE) FIELDS(#EMPNO #SALARY)
DEFINE FIELD(#HEADING1) TYPE(*CHAR) LENGTH(079) INPUT_A
DEFINE FIELD(#HEADING2) TYPE(*CHAR) LENGTH(079) INPUT_A
DEFINE FIELD(#HEADING3) TYPE(*CHAR) LENGTH(079) INPUT_A
DEF_COND NAME(*AS400) COND('*CPUTYPE = AS400')

```

```

CHANGE FIELD(#HEADING1) TO("EXAMPLE 1: Select all employees v
CHANGE FIELD(#HEADING2) TO(*BLANKS)
CHANGE FIELD(#HEADING3) TO("This can be done with one SELECT_

```

```

BEGIN_LOOP
CHANGE FIELD(#HEADING1) TO("EXAMPLE 1: Select all employees v
CLR_LIST NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#EMPNO #SALARY)
        USING('SELECT "EMPNO", "SALARY", FROM "XDEMOLIB"."PSI
                WHERE "SALARY" BETWEEN 30000 AND 60000')
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT

```

```
EXECUTE SUBROUTINE(DISP)
CHANGE FIELD(#HEADING1) TO("EXAMPLE 2: Select all employees v
CLR_LIST NAMED(#EMPBROWSE)
SELECT_SQL FIELDS(#EMPNO #SALARY)
        USING('SELECT "EMPNO", "SALARY", FROM "XDEMOLIB"."PSI
                WHERE "SALARY" NOT BETWEEN 30000 AND 60000')
ADD_ENTRY TO_LIST(#EMPBROWSE)
ENDSELECT
EXECUTE SUBROUTINE(DISP)
END_LOOP
```

```
SUBROUTINE NAME(DISP)
DISPLAY FIELDS(#HEADING1 #HEADING2 #HEADING3) DESIGN(*I
ENDROUTINE
```

For more examples of the SELECT_SQL command please see 'All About SELECT_SQL' in The Set Collection.

8.19.3 SELECT_SQL Free Format References

SAA Structured Query Language/400 Reference (SC41-9608)
SAA Structured Query Language/400 Programmers Guide (SC41-9609)

8.20 SET

The SET command sets a property in a component

Also See

[8.20.1 SET Parameters](#)

[8.20.2 SET Examples](#)

Required

```
SET ---- COM -----Component Name -----  
>  
  
>---- Property Values -----|
```

8.20.1 SET Parameters

COM

Property and Value

COM

COM specifies the component. You can define up to 50 components.

Property and Value

The property and the value to which it is to be set. Note that the value can also be the current value of another component.

8.20.2 SET Examples

This code sets the caption of #MyButton to "OK":

```
SET COM(#MyButton) Caption('OK')
```

This code disables buttons #PHBN_1 and #PHBN_2:

```
set COM(#phbn_1 #phbn_2) Enabled(false)
```

This code sets the ColumnWidth of the first grid in an ActiveX grid to 50:

```
SET COM(#Com_Grid) ColumnWidth<1>(50)
```

This code sets the caption of a label to the current value of the #SURNAME field on the form:

```
SET COM(#label_1) Caption(#SURNAME)
```

Note that if a field is used in a list-type component in a form, its current value can be passed using the field name exactly as if the actual field was on the form. So the code above would work if the form contained a list-type component which has a column with the Source property set to #SURNAME.

8.21 SET_REF

The SET_REF command is used to assign a component reference to a variable that has been defined with a dynamic reference in the DEFINE_COM statement. With a dynamic reference, an instance of a component is created only when a reference to it is assigned with the SET_REF command.

Note: Beware of using long object names with this command.
SET_REF Com(#MyObject) To(*CREATE_AS
#MyFormWithLongNames) works with long names.
SET_REF Com(#MyObject) To(*CREATE_FROM
#MyFormWithLongNames) doesn't work as it is runtime and long
names are not available.

Also See

[8.21.1 SET_REF Parameters](#)

[8.21.2 SET_REF Examples](#)

[8.13 IF_REF](#)

Required

```
SET_REF --- COM ----- Variable name|variable name-----  
-->  
  
  >-- TO ----- Reference expression ----->  
  
  >-- CONTEXT --- *DEFAULT -----  
>  
    *OWNER  
    *MODULE  
    *NEW  
    *NAMED ---- #Variable  
  
  >-- COM_ERROR--- *ABORT -----  
|  
    *IGNORE
```

**SET_NULL*

8.21.1 SET_REF Parameters

COM

TO

CONTEXT

COM_ERROR

COM

The COM parameter specifies the component or components to which the result of the reference expression specified in the TO parameter is assigned. The components specified in this parameter must have been specified with a dynamic reference in the DEFINE_COM parameter.

TO

Note: SET_REF *create only works with object short names. It cannot load objects using their long names because in runtime long names are not available.

The TO parameter derives a component reference which is assigned to the variables named in the COM parameter. The reference expression has to be in one of the following forms:

#VariableName	Assigns the reference contained in the variable name to each of the variables identified by the COM parameter. The variables must be compatible.
*NULL	Assigns a *NULL reference to all the component variables identified by the COM parameter.
*DYNAMIC #VariableName	Assigns the reference contained in the variable name to each of the variables identified by the COM parameter. No compile time checking is performed to see if the variables are compatible. At run-time, if the variable name cannot be cast to any of the variables identified by the COM parameter, an error is raised.
*CREATE_AS ClassName	Creates an instance of the component identified by the class name. The class must be defined in the LANSAs repository,

	<p>for instance #SALARY.Visual.</p> <p>Note that if you create an instance of a keyed collection you must use the parameterized version of the collection syntax: SET_REF #variable To(*Create_As #Prim- KCol<#PRIM_PHBN #STD_NUM>)</p>
*CREATE_FROM ClassName	<p>Creates an instance of the component identified by the class name. The class name can be either a string or a variable.</p> <p>At run-time, if the class name cannot be resolved, an error is raised.</p>

CONTEXT

The CONTEXT parameter controls how the new component reference is assigned a context in which it can obtain memory. This parameter enables management of object pooling.

By default the memory usage of objects is managed automatically by loading/unloading of a DLL or by passing of an component from one form to another. Memory is removed when no more objects are active in the pool.

The CONTEXT parameter allows you to control the memory pooling of objects programmatically, for instance you can load related objects into their own memory area.

For example you may load a large and complex form into memory then close the form but a single component (or an instance of an object) in the form may still be active and therefore the memory is not released. The CONTEXT parameter can be used to manage that form by creating a separate pool for the remaining object so that the main memory can be released.

The CONTEXT parameter can only be used when the TO parameter is set to *CREATE_AS or *CREATE_FROM. It can have one of these values:

*DEFAULT	For primitive component classes *DEFAULT is *OWNER. For non-primitive component classes it is *MODULE.
*OWNER	Use the context of the instance of the component for which the routine is executing.
*MODULE	Use the context allocated to the module (DLL) that contains the

	routine that is executing the SET_REF command.
*NEW	Create a new context for this component.
*NAMED #Variable	The context is the component reference identified by the variable name.

COM_ERROR

The COM_ERROR parameter indicates the action to be taken when a SET_REF assignment fails. The assignment can fail because:

- It is unable to find the component DLL for the name supplied for the *CREATE_FROM/*CREATE_AS options.
- It is unable to cast the component to the class required by the variable supplied to the SET_REF's COM() parameter

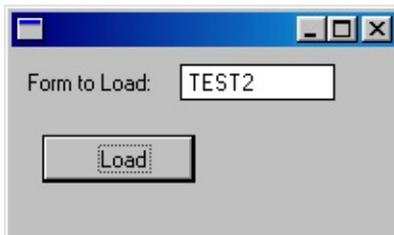
The COM_ERROR parameter can have these values:

*ABORT	The component execution is aborted if the SET_REF assignment fails.
*IGNORE	The failed SET_REF assignment is ignored. *IGNORE does not change the value of the reference variable.
*SET_NULL	The failed SET_REF assignment is ignored. A *NULL reference is set to the variable supplied in the SET_REF's COM() parameter.

8.21.2 SET_REF Examples

Example 1: *CREATE_FROM

This form can be used to load (create and display) instances of other forms:



This loader form has a collection of forms (#FORMS). When you click the Load button, a new instance of the form which you specified is created and added to the #FORMS collection and then shown.

```
BEGIN_COM HEIGHT(123) LEFT(296) TOP(120) WIDTH(209)
DEFINE_COM CLASS(#PRIM_PHBN) NAME(#PHBN_1) CAPTION('Load
*The name of the form to be loaded
DEFINE_COM CLASS(#STD_OBJ.Visual) NAME(#STD_OBJ) CAPTION('
*The form counter and collection
define #FormTot Reffld(#STD_NUM) default(0)
DEFINE_COM CLASS(#PRIM_KCOL) NAME(#FORMS) COLLECTS(#PR

define #Position Reffld(#STD_NUM) default(1)

EVTROUTINE HANDLING(#PHBN_1.Click)
* increase form counter by one
Change #FormTot '#FormTot + 1'

* create a new instance of the named form and set a reference to it
Set_Ref #Forms<#FormTot> (*Create_from #Std_Obj.Value)

* control the position of the form
Set #Forms<#FormTot> Left(#Position) Top(#Position)
Change #Position '#Position + 10'

*show the form
Invoke #Forms<#FormTot>.ShowForm
ENDROUTINE
```

END_COM

Try copying and pasting the source code to a form component and compile it. Execute the form and use it to create and display instances of other forms by specifying the name of the form to be displayed and clicking the Load button.

Example 2: *CREATE_AS

The component reference to an ActiveX component for Microsoft Word is defined dynamically so that no reference is created when the component definition statement is executed.

```
DEFINE_COM Class(#VA_WORD.Application) Name(#WordApp) Reference
```

A reference to the object is created only when it is explicitly declared in a SET_REF command.

```
SET_REF COM(#WordApp) To(*CREATE_AS #VA_WORD.application)
```

Because the reference is created with the *CREATE_AS keyword, a new instance of the #WordApp component is created.

Example 3: *DYNAMIC and Variable Name

A variable of the type Microsoft Word document is defined so that the this variable can be used to refer to a document:

```
DEFINE_COM Class(#VA_WORD.Document) Name(#WordDoc) Reference(
```

The variable name is used to create a reference to a newly created document which is returned (in the parameter add_retval) when a document is created:

```
INVOKE Method(#WordApp.documents.add) add_retval(#WordDoc)
```

This reference is then used to access the methods, properties and events of the document:

```
Set Com(#edit_1) Value(#WordDoc.Name_COM)  
INVOKE Method(#WordDoc.PrintOut)
```

Later on the same variable is changed to refer to which ever document is active at the moment:

```
SET_REF COM(#WordDoc) To(#WordApp.ActiveDocument)
```

8.22 SIGNAL

The SIGNAL command triggers a user-defined event in a form. The form must contain a DEFINE_EVT command for defining the event.

Custom-defined events are used in multi-form applications to communicate events to an owner form.

Also See

[8.22.1 SIGNAL Parameters](#)

[8.22.2 SIGNAL Examples](#)

[8.5 DEFINE_EVT](#)

Required

SIGNAL ---- *EVENT* ----- *Event Name* -----
>

Optional

>---- *Parameter and Value* -----|

8.22.1 SIGNAL Parameters

EVENT

Parameter and Value

EVENT

EVENT specifies the component and event being triggered.

Parameter and Value

If the event being signaled has an output parameter (specified with a DEFINE_MAP statement in the DEFINE_EVT block), the value for this parameter is specified in the SIGNAL command.

8.22.2 SIGNAL Examples

This statement signals a custom-defined Button_1_Clicked event.

```
SIGNAL Event(Button_1_Clicked)
```

This statement signals that the Employee_Created event has been triggered and it passes the employee number of the new employee to an owner form.

```
SIGNAL Event(employee_created) this_emp(#empno)
```

An owner form can respond to this event using an event routine:

```
EVTROUTINE Handling(#FormB.employee_created) this_emp(#TheValue)  
Set Com(#empno) Value(#TheValue)  
ENDROUTINE
```

8.23 WEB_MAP

Once you have declared a WEBROUTINE, you can map its incoming and outgoing fields and lists by using the WEB_MAP statement. A WEBROUTINE can have multiple WEB_MAP statements.

In addition to specifying WEB_MAPs inside WEBROUTINE blocks, you are allowed to declare WEB_MAPs inside a BEGIN_COM block of a WAM. This technique allows you to map fields and lists into every WEBROUTINE in your WAM without having to explicitly define WEB_MAPs in each WEBROUTINE.

Also See

[8.23.1 WEB_MAP Parameters](#)

[8.23.2 WEB_MAP Examples](#)

[8.24 WEBROUTINE](#)

Required

```
WEB_MAP ----- *INPUT -----  
>  
    *OUTPUT  
    *BOTH  
    *NONE  
>---FIELDS----- field names  attributes----->  
    list name
```

Optional

```
>-- OPTIONS ---- *PERSISTS -----  
>
```

8.23.1 WEB_MAP Parameters

FIELDS

FOR

OPTIONS

FOR

The FOR parameter may have one of the following values:

*INPUT	Incoming data
*OUTPUT	Outgoing data
*BOTH	Both incoming and outgoing data
*NONE	Neither incoming nor outgoing data, used to declare session state.

If an attribute is not specified, a default of *INPUT is assumed.

When a FOR(*NONE) declaration is used, the declared fields and lists will not be mapped in or out of WEBROUTINES. If OPTIONS(*PERSIST) is specified with FOR(*NONE), the values of the fields and lists so declared will be maintained as part of session data. Declaring fields and lists this way at the WAM level (that is, after BEGIN_COM) will make the values of those fields and lists available for all WEBROUTINES in the WAM and will be maintained as part of session data (that is, spanning multiple WEBROUTINE executions).

FIELDS

Specifies the names of the field(s) or names of the lists which are used in the WEB_MAP.

All fields nominated in this parameter can be defined in the LANSAs Repository or via a DEFINE FIELD command in the WAM.

Fields may have the following attributes assigned to them:

- *INPUT - a field, which accepts input (i.e. an input box for HTML Technology Service)
- *OUTPUT - a field, which displays output only
- *HIDDEN - a field, which contains a value, but is hidden on display

- ***PRIVATE** - a value for this field or list is available, but XSL generator does not generate it into the XSL Stylesheet. This is useful for fields or lists, which are used for purposes other than to be directly displayed or accept input. For example, a list that contains entries for a dropdown box can be used by the dropdown weblet. However, a browse list table for it will not be generated if it is marked with this attribute.
- ***INLINE** – Generate the list using inline parsing instead of XSLT. Not required if the WAM default has been set to **INLINE** with the **INLINE** parameter of **BEGIN_COM**.
- ***NOINLINE** – Generate the list using XSLT instead of inline parsing. Only required if the WAM default has been set to **INLINE** with the **INLINE** parameter of **BEGIN_COM**.

OPTIONS

The **OPTIONS** parameter may have the following values:

*PERSIST	Declares that fields and lists are to retain or persist data so that it is available across WEBROUTINE executions for the duration of a Web session
-----------------	--

In addition to declaring an **OPTIONS(*PERSIST)**, the **WEBROUTINE** needs to have an initial **SessionStatus** set to **Active**. This is done by, either setting it for the **WAM** or by setting **OnEntry(*SessionStatus_Active)** keyword for individual **WEBROUTINE**. By setting **SessionStatus** to **Active**, in this way, ensures that the **WEBROUTINE** will load the session state before it starts execution.

8.23.2 WEB_MAP Examples

In this example, the SearchQuery WebRoutine will display a page to allow a user to request a search. The fields are *OUTPUT only as no values are sent back to this Webroutine. The next Webroutine (Browse) will specify #SURNAME as in input. It requires #SURNAME, #STDREENTRY, and #STD_COUNT to be able to retrieve the next or previous.

```
WEBROUTINE NAME(SearchQuery) DESC('Search Criteria')
WEB_MAP FOR(*OUTPUT) FIELDS(#SURNAME (#STDREENTRY *HIDE

ENDROUTINE
```

The Browse WebRoutine will allow the user to browse through the list of employees (#EMPLISTPG) a page at a time. It requires #SURNAME, #STDREENTRY, and #STD_COUNT to be able to retrieve the next or previous.

```
WEBROUTINE NAME(Browse) DESC('Browse Employees')
WEB_MAP FOR(*BOTH) FIELDS((#SURNAME *HIDDEN) (#STD_COU
WEB_MAP FOR(*OUTPUT) FIELDS(#EMPLISTPG)

ENDROUTINE
```

Finally, the Details WebRoutine retrieves employee's details for display to user. The #EMPNO and #SURNAME are both sent and received, but displayed as output fields only. The #STDREENTRY field is used to communicate status with other Webroutines.

```
WEBROUTINE NAME(Details)
WEB_MAP FOR(*OUTPUT) FIELDS((#GIVENAME *OUTPUT) (#ADDRI
WEB_MAP FOR(*BOTH) FIELDS((#EMPNO *OUTPUT) (#SURNAME *C

ENDROUTINE
```

8.24 WEBROUTINE

A WEBROUTINE is an entry point into a WAM. A WAM may contain one or more WEBROUTINEs. The WEBROUTINE names have to be unique within a WAM only. It is possible to have more than one WAM with a WEBROUTINE with the same name.

You can add a new WEBROUTINE anywhere inside BEGIN_COM block in your WAM component by using a WEBROUTINE keyword.

The WEBROUTINE command is followed by a one or more WEB_MAP commands - each WEB_MAP command identifying fields or a lists that are to be mapped as input or output of the Webroutine. Collectively, the WEB_MAP commands define all the inputs and outputs of a Webroutine. All fields and lists defined FOR(*INPUT) can be sent to the called WEBROUTINE and they will be mapped into those fields and lists. Any fields or lists defined FOR(*OUTPUT) are outgoing from the WEBROUTINE and can be visualized on the output page. In addition field and list mapping can be bidirectional in which case FOR(*BOTH) specification should be used.

A WEBROUTINE is invoked via a URL request and is uniquely identified by the key of Web Application Module (up to 10 characters) and WEBROUTINE name (up to 20 characters).

Each WEBROUTINE generates separate XML and XSL data. The XML data contains all the fields and lists nominated by the WEB_MAP FOR(*OUTPUT) or FOR(*BOTH) commands.

Also See

[8.24.1 WEBROUTINE Parameters](#)

[8.24.2 WEBROUTINE Examples](#)

Required

WEBROUTINE ----NAME -----Name of WebRoutine -----
----->

Optional

```
>-- DESC ----- Description of routine ----->

>-- RESPONSE ----*DEFAULT -----
>
    *NONE
    *JSON
    Response Variable (LOB)

>-- OPTIONS ----*METADATA -----
>

>-- ONENTRY -----*SESSIONSTATUS_OF_WAM-----
----->
    *SESSIONSTATUS_NONE
    *SESSIOSTATUS_ACTIVE

>-- HELP -----Help Text ----->

>-- ServiceName- Name of Service -----
|
```

8.24.1 WEBROUTINE Parameters

DESC

HELP

NAME

ONENTRY

OPTIONS

RESPONSE

SERVICENAME

NAME

NAME is the unique name of the WebRoutine. The name must be unique in the WAM. The name can be up to 20 characters long.

DESC

Use the DESC parameter to write a brief description for the WebRoutine. It can be 40 characters long. By default, this description will be displayed on the output page.

RESPONSE

Use the RESPONSE parameter to indicate a response type other than the default, which is to send a result document.

Response type can be:

*DEFAULT: Send a result document (for example, an XHTML page)

*NONE: The webroutine produces no output

*JSON: The webroutine's web maps (fields and lists) is sent as a JSON response with MIME type application/json and encoded in UTF-8.

Response variable name: Enter the name of the variable for a LOB response.

See [LOB Data Types and Stream Files](#) in the Web Application Module (WAM) Guide for details.

OPTIONS

Only valid for RESPONSE(*JSON). Use the OPTIONS parameter to indicate whether the JSON response should include captions.

*METADATA: Include field/list captions in JSON response.

HELP

Not implemented.

SERVICENAME

A Service Name is unique to all WAMs, i.e. a Service Name can only be used once in a LANSAs Partition. Once deployed, the WEBROUTINE may be invoked from the browser by providing just the Service Name and Partition keyword in the URL (without the additional keywords). A Partition keyword may be omitted if LANSAs for the Web has been setup to always run in a configured partition.

Using a Service Name provides greater flexibility when deploying WAM applications. For example, it allows applications to be re-deployed to a different Partition, WAM or WEBROUTINE without having to modify any external URL references to it.

ONENTRY

Is used to override the SessionStatus property setting for individual WEBROUTINEs.

This is useful when the default SessionStatus is Active, which prevents execution of WEBROUTINEs in the WAM unless a session is created. In this situation, you need at least one WEBROUTINE that can be executed initially so that a session can be created. ONENTRY Parameter value of *SESSIONSTATUS_NONE can be used to turn off session validation and session data loading for that WEBROUTINE

Can be one of:

*SESSIONSTATUS_NONE does not validate the session and will not load any session data when the WEBROUTINE is entered.

*SESSIONSTATUS_OF_WAM uses the value specified by WAM SessionStatus property.

*SESSIONSTATUS_ACTIVE enables validation of session and will load session data, if the session is valid, when the WEBROUTINE is entered.

The default value is *SESSIONSTATUS_OF_WAM.

8.24.2 WEBROUTINE Examples

For example, a WAM can be created to allow a user to search for an employee in a file, select the employee from a search list and display the employee details.

Three WebRoutines might be defined as follows:

```
FUNCTION OPTIONS(*DIRECT)
BEGIN_COM ROLE(*EXTENDS #PRIM_WAM)
```

```
WEBROUTINE NAME(SearchQuery)DESC('Search Criteria')
ENDROUTINE
```

```
WEBROUTINE NAME(Browse)DESC('Browse Employees')
ENDROUTINE
```

```
WEBROUTINE NAME(Details)DESC('Details') SERVICENAME(EmpDetail)
ENDROUTINE
```

```
WEBROUTINE NAME(Send_Sample) DESC('Sample Document')
RESPONSE(#HTTPR)
ENDROUTINE
```

```
WEBROUTINE NAME(Update) DESC('JSON response')
RESPONSE(*JSON)
END_COM
```

The Details WebRoutine can be invoked from the browser by providing just the Service Name keyword in the URL (without the additional keywords):

```
http://localhost/cgi-bin/lansaweb?srve=EmpDetails
```

Refer to the [8.23.2 WEB_MAP Examples](#) for example of how data could be mapped for each WebRoutine.

8.25 Component Variables and Values

[8.25.1 Referring to Property Values](#)

[8.25.2 Com_Owner, Com_Ancessor and Com_Self - Generic References to Components](#)

[8.25.3 Variant Variable](#)

[8.25.4 Qualified Properties](#)

[↑ 8. RDMLX Commands and RDMLX Features](#)

8.25.1 Referring to Property Values

When you refer to property values and event names you must always qualify them by the name of the component they belong to.

You can assign the value of a property to another component. For instance this statement assigns the value of the Level property of the current item in a tree view to a field called #level.

```
Change #LEVEL #Trvw_1.CurrentItem.Level
```

Often in your code you want to test the value of a property. You do this using the IF statement where the condition is specified using the property name qualified by the name of the component, a standard operator and the value of the property:

```
IF COND('#rdbn_3.ButtonChecked *eq True')  
  Set Com(#SPOUSE #MARRIED #DIVORCED) Enabled(true)  
ENDIF
```

Or similarly:

```
IF COND('#lvcl_1.SortDirection *eq Ascending')  
  Set Com(#lvcl_1) SortDirection(Descending)  
ENDIF
```

[↑ 8.25 Component Variables and Values](#)

8.25.2 Com_Owner, Com_Ancessor and Com_Self - Generic References to Components

Com_Owner

Com_Owner is commonly used throughout RDMLX and is a variable that always refers to the object in which the command is specified.

So, when adding controls to a component using the designer, or implementing events and method routines, any source code generated by the editor will refer to the component currently being edited using the generic name #COM_OWNER.

```
Define_Com Class(#PRIM_LABL) Name(#Label) Parent(#COM_OWNER)
```

OR

```
EvtRoutine Handling(#Com_Owner.Initialize)
```

```
#Com_owner.Prepare
```

```
Endroutine
```

As a large percentage of code typically needs to refer to the current location, the use of the generic Com_owner to mean 'here' greatly simplifies code understanding. It also means that copied code can be pasted directly in to a target without the need to update component references with specific class names.

For relatively simple applications that don't use inheritance beyond a single level, there is no reason to move away from the use of Com_Owner. However, where multiple layers of inheritance are being used, Com_Ancessor and Com_Self can be used to execute code defined in the different layers of the inheritance hierarchy.

Com_Ancessor

Com_Ancessor refers directly to the class specified as the ancestor of the current component.

Thus, when in an application that uses inheritance, we might define basic behavior in a base ancestor class but override that behavior by redefining the method in the inheriting class. At execution time the processing in the ancestor class will be ignored and the redefined method will be executed.

```
Mthroutine Name(Prepare) Options(*Redefine)
```

```
* Class specific processing
```

```
Endroutine
```

Whilst this typical use of a redefined method is often desirable, there may also be circumstances where rather than wanting to completely change the processing, we simply need to augment it. To enable this we might write the following

```
Mthroutine Name(Prepare) Options(*Redefine)
```

```
#Com_Ancessor.Prepare
```

```
* Class specific processing
```

```
Endroutine
```

Here, the method is redefined but the very first line of the new method causes the ancestor version of the method to be executed. This technique is typically used when a change to the ancestor processing is the exception rather than the rule.

Com_Self

Com_Self and Com_owner are very similar in use apart from one very distinct difference. While Com_owner always refers to the current component, Com_Self refers to the current component taking into consideration any redefined methods further up the inheritance chain.

In the previous Com_ancestor example, redefining the method represented the exceptional case, but in a scenario where it is commonplace, the need to remember to execute ancestor code can cause complications. In this situation it is better to drive the processing from the ancestor class.

Thus in the Ancestor we might have something similar to the following.

```
Mthroutine Name(Prepare)
```

```
* Run base code then class specific code
```

```
#Com_owner.PrepareBase
```

```
#Com_Self.PrepareSelf
```

```
Endroutine
```

```
Mthroutine Name(PrepareBase) Options(*Final)
```

```
* Base class processing
```

```
* Final – This method cannot be redefined
```

```
Endroutine
```

```
Mthroutine Name(PrepareSelf)
```

```
* Redefine in inheriting classes
```

```
Endroutine
```

In the inheriting class we would then see:

```
Mthroutine Name(PrepareSelf) Options(*Redefine)
```

```
* Class specific processing here
```

```
Endroutine
```

At runtime, invoking the Prepare method would result in the PrepareBase method being executed. However, because the invocation of PrepareSelf uses Com_Self, the runtime looks further up the inheritance chain for a redefined version and executes that rather than the version of the method defined in the ancestor.

Unfortunately, it is all too easy to accidentally use Com_owner rather than Com_Self and a common issue found during development is that redefined code is never actually executed because Com_owner has been used instead of Com_Self.

[↑ 8.25 Component Variables and Values](#)

8.25.3 Variant Variable

You can use the Variant component class #PRIM_VAR to create a variant component variable. A variant component variable can contain any type of data (strings, integers, decimals, booleans, components).

A variant component has properties for testing the kind of value contained by the component. You can retrieve its value in converted forms like numbers, strings or booleans.

Variants make it possible for components to give access to values whose type cannot be statically determined. For example, the value of a cell in a grid cannot be statically defined - it depends on the type of the cell's column. By returning a variant, the grid control can provide access to the cell's value without causing compiler errors about the declared type of the value. You then need to write the program in a way that understands what type(s) are in the grid.

Variants are also used in dynamic programming. There are situations (especially with ActiveX) where the type information of a component cannot be determined at compile time. It must be processed at runtime. When using variants, the Visual LANSA compiler does not attempt to resolve the method or property because it has no idea what type of component is stored in the variant. It must wait until runtime to resolve all this information. This behavior enables the an ActiveX control to supply a component of unknown type and only when the component is called will the type information be requested.

This statement defines a variant variable called #MYVARIANT:

```
Define_Com Class(#PRIM_VAR) Name(#MYVARIANT)
```

You can assign a value to the variant variable either using its Value property (in which case the type of the value is of unspecified type):

```
Set Com(#myvariant) Value(#XYZ)
```

Or by explicitly assigning its value type using the String, Integer, Boolean, Decimal or Component properties of the variable:

```
Set Com(#myvariant) Integer(#XYZ)
```

Explicitly assigning the type of value is necessary only when you use the variable in RDML commands, in RDMLX the type can be unspecified.

When you read the value of the variable it is automatically converted to the type

of the field that receives the value:

```
Set Com(#Out_INTEGER) Value(#myvariant)
```

Or you can also explicitly specify the type of value:

```
Set Com(#Out_INTEGER) Value(#myvariant.Integer)
```

To ensure that the value in the variant is one that you can support you use the `ValueType` property of the variable.

```
IF Cond('#myvariant.ValueType = VarInteger')
Set Com(#Out_INTEGER) Value(#myvariant)
ELSE
Set Com(#Out_INTEGER) Value(0)
ENDIF
```

If this check is not performed and the value cannot be converted at runtime, you will get a runtime error.

The `ValueType` property is an enumeration whose value can be one of the following symbols:

- `varNull`
- `varEmpty`
- `varInteger`
- `varDouble`
- `varString`
- `varDecimal`
- `varBoolean`
- `varComponent`

[↑ 8.25 Component Variables and Values](#)

8.25.4 Qualified Properties

Many ActiveX components use properties with qualifying arguments. The syntax for a qualified property is:

```
Component.property<qualifier>
```

For example this is how you would refer to a qualified ColumnWidth property which sets the width of a particular column in the ActiveX grid:

```
#Com_Grid.ColumnWidth<1>
```

You can retrieve the value this property the same way as you would a non-qualified property:

```
IF '#Com_Grid.ColumnWidth<1> *GT 100'  
* Some code  
ENDIF
```

Similarly, you can set the value of a qualified property the same way as would a non-qualified property:

```
SET COM(#Com_Grid) ColumnWidth<1>(50)
```

You can use a field as the qualifier. In this example the qualifier of the column is set by the field #STD_NUM:

```
SET COM(#Com_Grid) ColumnWidth<#STD_NUM>(50)
```

A property can have more than one qualifier. For example a Cell property of an ActiveX grid can have two qualifying arguments (referring to row and column) to indicate the position of a cell in the grid:

```
#Com_Grid.Cell<1,1>
```

[↑ 8.25 Component Variables and Values](#)

8.26 Function Libraries

An alternative way of using [Intrinsic Functions](#) is to import function libraries. Function Libraries are a component that contains a set of routines defined using the MthRoutine command. These function library routines can be used in expressions.

There are function libraries for the following:

string #PRIM_LIBS

number #PRIM_LIBN

date and time #PRIM_LIBD

*VARIANT #PRIM_LIBV

For string, number, date and time, using [Intrinsic Functions](#) is simpler than using a function library.

To handle objects of type *VARIANT, you need to use the variant function library #PRIM_LIBV. (Refer to [8.27 Variant Handling](#).)

When using function library methods the syntax is different.

For example the Uppercase method:

```
#Subject.Uppercase()
```

is specified as:

```
Uppercase( Subject )
```

Function libraries are introduced into a RDMLX object by the [8.14 IMPORT](#) command which is specified immediately after the FUNCTION statement. For example:

```
Function Options(*DIRECT)
* Import the variant library #prim_libv.
Import Libraries(#PRIM_LIBV)
```

Once a Function library has been imported, the routines defined in the library can be used in expressions.

↑ 8. RDMLX Commands and RDMLX Features

8.27 Variant Handling

A Function Library is a component that contains a set of routines defined using the MthRoutine command. (Refer to [8.26 Function Libraries](#).) These function library routines can be used in expressions. To handle objects of type *VARIANT, you need to use the variant function library #PRIM_LIBV. (For string, number, date and time, using [Intrinsic Functions](#) is simpler than using a function library.)

Variants can contain any type of data (strings, integers, decimals, booleans, components). You can use variant functions for testing the kind of value contained in the variable and you can retrieve its value in converted forms like numbers, strings or booleans.

Variants make possible the generic processing of values regardless of their type. For example, the value of a grid cell cannot be known by the compiler before the application is executed. Therefore the Value parameter of grid EditorChanged and ItemChangedAccept events returns the value in the cell as a variant so that compiler errors about the declared type of the value do not occur. You then need to write the program in a way that understands what type(s) are in the grid.

Also the EditorChanged and ItemChangedAccept events of tree and list views return the Value parameter as a variant. Similarly, many ActiveX controls return and accept values as variants. You can also use variants in your own dynamic programs for generic processing of values regardless of their type. For example:

```
Define_Com Class(*VARIANT) Name(#lclVariant)
```

Using the *VARIANT class is the recommended approach. Alternatively, you can use primitive variant component #PRIM_VAR and its properties and methods.

#PRIM_LIBV supports these functions:

[8.27.1 VarAsBoolean](#) [8.27.8 VarIsNull](#)

[8.27.2 VarAsDecimal](#) [8.27.9 VarIsNullReference](#)

[8.27.3 VarAsInteger](#) [8.27.10 VarIsNumber](#)

[8.27.4 VarAsReference](#) [8.27.11 VarIsReference](#)

[8.27.5 VarAsString](#) [8.27.12 VarIsString](#)

[8.27.6 VarIsBoolean](#) [8.27.13 VarType](#)

[8.27.7 VarIsEmpty](#)

Also See

[8.26 Function Libraries](#)

[Intrinsic Functions](#)

[↑ 8. RDMLX Commands and RDMLX Features](#)

8.27.1 VarAsBoolean

VarAsBoolean returns the Subject variant as a Boolean.

VarAsBoolean(Subject)

[↑ 8.27 Variant Handling](#)

8.27.2 VarAsDecimal

VarAsDecimal returns the Subject variant as a decimal number.

VarAsDecimal(Subject)

[↑ 8.27 Variant Handling](#)

8.27.3 VarAsInteger

VarAsInteger returns the Subject variant as an integer.

VarAsInteger(Subject)

[↑ 8.27 Variant Handling](#)

8.27.4 VarAsReference

VarAsReference returns the Subject variant as a component reference.

VarAsReference(Subject)

[↑ 8.27 Variant Handling](#)

8.27.5 VarAsString

VarAsString returns the Subject variant as a string.

VarAsString(Subject)

[↑ 8.27 Variant Handling](#)

8.27.6 VarIsBoolean

VarIsBoolean Returns a Boolean True if the Subject variant can be converted to a Boolean, otherwise it returns a Boolean False.

```
VarIsBoolean( Subject )
```

Example

This property routine tests if the variable is a Boolean:

```
Ptyroutine Name(Set_uSignalSelection)
Define_Map For(*input) Class(*variant) Name(#lcVariant)

If Cond(VarIsBoolean( #lcVariant ) *EQ True)
Execute Subroutine(FP_SETB) With_Parms(#USE_NAME uSignalSelection 1
Else
Execute Subroutine(FP_SET) With_Parms(#USE_NAME uSignalSelection 1 #
Endif
```

[↑ 8.27 Variant Handling](#)

8.27.7 VarIsEmpty

VarIsEmpty returns a Boolean True if the Subject variant does not contain a value, otherwise it returns a Boolean False.

```
VarIsEmpty( Subject )
```

Example

This statement checks if a value has been assigned to a grid cell:

```
If (VarIsEmpty( #grid_1.focusCell.column.EditorPAr ))  
Use Builtin(MESSAGE_BOX_SHOW) With_Args(OK OK Information 'Varia  
Endif
```

[↑ 8.27 Variant Handling](#)

8.27.8 VarIsNull

VarIsNull returns a Boolean True if the Subject variant contains the special Null value, otherwise it returns a Boolean False.

```
VarIsNull( Subject )
```

Example

This statement tests if the value in the grid cell is unknown or missing:

```
If (VarIsNull( #grid_1.focusCell.value ))  
Use Builtin(MESSAGE_BOX_SHOW) With_Args(OK OK Information 'Varia  
Endif
```

[↑ 8.27 Variant Handling](#)

8.27.9 VarIsNullReference

VarIsNullReference returns a Boolean True if the Subject variant contains a component reference and the reference is the special *NULL component reference, otherwise it returns a Boolean False.

VarIsNullReference(Subject)

[↑ 8.27 Variant Handling](#)

8.27.10 VarIsNumber

VarIsNumber returns a Boolean True if the Subject variant can be converted to a number, otherwise it returns a Boolean False.

```
VarIsNumber( Subject )
```

Example

This if statement tests if the value is a string:

```
If Cond(VarIsNumber( #TheValue ) *EQ True)
Use Builtin(MESSAGE_BOX_SHOW) With_Args(OK OK Information 'Varia
Endif
```

[↑ 8.27 Variant Handling](#)

8.27.11 VarIsReference

VarIsReference Returns a Boolean True if the Subject variant is a component reference, otherwise it returns a Boolean False.

VarIsReference(Subject)

[↑ 8.27 Variant Handling](#)

8.27.12 VarIsString

VarIsString returns a Boolean True if the Subject variant can be converted to a string, otherwise it returns a Boolean False.

```
VarIsString( Subject )
```

Example

This if statement tests if the value is a string:

```
If Cond(VarIsString( #TheValue ) *EQ True)
Use Builtin(MESSAGE_BOX_SHOW) With_Args(OK OK Information 'Varia
Else
Use Builtin(MESSAGE_BOX_SHOW) With_Args(OK OK Information 'Varia
Endif
```

[↑ 8.27 Variant Handling](#)

8.27.13 VarType

VarType returns the type of the value current stored in the Subject variant.

```
VarType( Subject )
```

Possible results are:

- VarNull
- VarEmpty
- VarInteger
- VarDouble
- VarString
- VarDecimal
- VarBoolean
- VarComponent

Example

This example checks whether the value returned from a grid is a string:

```
If Cond(VarType( #TheValue ) *EQ VarString)
  Use Builtin(MESSAGE_BOX_SHOW) With_Args(OK OK Information 'Varia
Endif
```

[↑ 8.27 Variant Handling](#)

8.28 Enhanced Expressions

In Full RDMLX, you can use [8.28.1 Expressions as Values](#) for many command parameters thus eliminating the need for additional commands to prepare the required value. For example you can specify:

```
MESSAGE MSGTXT('Message: name is ' + #fullname)
```

It is now possible to use component [8.28.2 Methods in Expressions](#). For example:

```
Change #STD_TEXT To(#COM_OWNER.StringMethod(#ADDRESS1, #AD
```

You can define one of the parameters of a method to return [8.28.3 Method Results](#).

You can use [8.28.4 Named Parameters](#) instead of referring to parameters by position:

```
#RESULT = #COM_OWNER.MethodOne( ParmTwo := 2 )
```

Other operators available:

[8.28.5 *Not Operator](#)

[8.28.7 *IsEqualTo and *IsOfType Operators](#)

[8.28.9 *ANDIF and *ORIF Logical Operators](#)

[8.28.6 *IS and *ISNOT Operator](#)

[8.28.8 *AS Operator](#)

[↑ 8. RDMLX Commands and RDMLX Features](#)

8.28.1 Expressions as Values

Full RDMLX supports the use of expressions in many parameters and properties.

EXECUTE Command

You can now use expressions as the value of the With_Parms parameter in the Execute command, thus eliminating the need for additional commands to prepare the required value. For example:

```
Execute Subroutine(Routine2) With_Parms(#STD_DESC ('B' + #Phbn_2.Capt
```

MSGTXT Parameter of Various Commands

The MSGTXT parameter can have as a value an expression which contains more than a single item or the expression is enclosed in parentheses. For example:

```
Message Msgtxt('Message: name is ' + #fullname)
```

```
Mthroutine Name(Trace)
```

```
Define_Map For(*INPUT) Class(#STD_TEXT) Name(#InTextOne)
```

```
Define_Map For(*INPUT) Class(#STD_TEXT) Name(#InTextTwo)
```

```
Message Msgtxt(#InTextOne.Value + " " + #InTextTwo.Value)
```

```
Endroutine
```

SET Command

The values assigned to properties selected on a SET command support an expression:

```
SET #COM_OWNER Left(#COM_OWNER.Left + 2) Top(#COM_OWNER.5
```

INVOKE Command

The values assigned to method parameters selected on an INVOKE command support an expression:

```
INVOKE #COM_OWNER.MethodOne ParmOne(#COM_OWNER.Left + 2) :
```

SIGNAL Command

The values assigned to event parameters selected on a SIGNAL command

support an expression:

```
SIGNAL EventOne ParmOne(#COM_OWNER.Left + 2) ParmTwo(#COM_O
```

SET_REF Command

You can use expressions in SET_REF commands:

```
Set_Ref #CurrentDepartment (*dynamic #COM_OWNER.dosomething(#ST
```

```
Set_Ref #CurrentDepartment (*dynamic #COM_OWNER.dosomething(iNu
```

```
MTHROUTINE NAME(dosomething )
```

```
DEFINE_MAP FOR( *INPUT ) CLASS( #STD_NUM) name(#iNumber)
```

```
DEFINE_MAP FOR(*RESULT) CLASS(#DEPTMENT) NAME(#oDepartme
```

```
Set_Ref #oDepartment (*create_As #DEPTMENT)
```

```
ENDROUTINE
```

[↑ 8.28 Enhanced Expressions](#)

8.28.2 Methods in Expressions

You can now use methods of a component and intrinsic functions in expressions. For example:

```
Change #STD_TEXT To( #COM_OWNER.StringMethod( #ADDRESS1, #AI
Change #STD_NUM To( #COM_OWNER.NumberMethod( #SALARY ) * 2 )
Change #STD_COUNT To( #ADDRESS1.Trim.CurSize)
```

The syntax is:

```
#VariableName[.Features].MethodName[ ( [Parameters] ) ]
```

When parameters are supplied to the method, the `MethodName` must be followed by the left parenthesis that starts the parameters. No imbedded spaces are allowed.

When no parameters are required, the parentheses are optional but if you supply a left parenthesis, it must follow the name of the method without any imbedded spaces.

Parameters can be specified by position or they can be [8.28.4 Named Parameters](#).

[↑ 8.28 Enhanced Expressions](#)

8.28.3 Method Results

Methods can have one of their *OUTPUT maps defined as *RESULT. In this way you can use a method in an expression as a function call that produces a factor of an expression.

For example:

```
Mthroutine Name(MakeMessage)
Define_Map For(*RESULT) Class(#STD_TEXT) Name(#OutTextOne)
Define_Map For(*INPUT) Class(#STD_TEXT) Name(#InTextOne)
Define_Map For(*INPUT) Class(#STD_TEXT) Name(#InTextTwo)
#OutTextOne := #InTextOne.Value + " " + #InTextTwo.Value
Endroutine
```

```
Evtroutine Handling(#PHBN_2.Click)
Begincheck
Datecheck Field(#STD_DATE) In_Format(*DDMMYY)
Endcheck Msgtxt(#COM_OWNER.MakeMessage( "#Phbn_2.Click", "Bad Da
Endroutine
```

Methods or intrinsic functions that return a Boolean result can be used in IF statements. This is similar in concept to the use of a defined condition (see DEF_COND command). However, because a method is being invoked, there are no limitations imposed upon the nature of the condition.

```
Mthroutine Name(Set_availability)

#Button1.Enabled := #COM_OWNER.Allow_Access
#SURNAME.readonly := *Not #COM_OWNER.Allow_Access
```

```
Endroutine
```

```
Mthroutine Name(Allow_Access)
Define_Map For(*RESULT) Class(#Prim_boln) Name(#RESULT)
```

```
#RESULT := True
```

```
If (#GIVENAME.Contains( 'ABC' ))
#RESULT := False
```

Endif

Endroutine

[↑ 8.28 Enhanced Expressions](#)

8.28.4 Named Parameters

All parameters of methods, keyed properties and function library routines are named. When invoking such a routine as an operation in an expression, usually you would code:

```
Invoke #COM_OWNER.MethodOne ParmOne(1) ParmTwo(2) ParmThree(#R
```

The same result could be achieved using an assignment statement and the following syntax:

```
#RESULT = #COM_OWNER.MethodOne( 1, 2 )
```

If ParmOne is optional and you want to use the default, you can simply pass a value for ParmTwo without specifying a positional value for parameter ParmOne by specifying ParmTwo by name:

```
#RESULT = #COM_OWNER.MethodOne( ParmTwo := 2 )
```

```
#RESULT = #COM_OWNER.MethodOne( ParmTwo := ((#A + #B) * #C) )
```

[↑ 8.28 Enhanced Expressions](#)

8.28.5 *Not Operator

*Not can be used to test or set the reciprocal of a Boolean result.

For example:

```
If (*Not #Object.Boolean)
```

or

```
#button.enabled := *not #button.enabled
```

Where complex logic processing is required to define a "good" or "bad" result, a good technique is to encapsulate the code in a method that returns a Boolean. Consequently, *Not can be used to perform processing based on the reciprocal result.

```
Evtroutine Handling(#STD_NUM.Changed)
If (*Not #COM_OWNER.IsValidEntry( #STD_NUM ))
* Do something
Endif
Endroutine
```

```
Mthroutine Name(IsValidEntry)
Define_Map For(*input) Class(#STD_NUM) Name(#NUMBER)
Define_Map For(*RESULT) Class(#prim_boln) Name(#RESULT)
#RESULT := False
If ((#NUMBER > 100) *And (#NUMBER < 200))
#RESULT := true
Endif
If ((#NUMBER > 300) *And (#NUMBER < 400))
#RESULT := true
Endif
Endroutine
```

[↑ 8.28 Enhanced Expressions](#)

8.28.6 *IS and *ISNOT Operator

You can use the *IS and the *ISNOT operators to perform type testing of component reference variables. The following code fragment illustrates the syntax of these operators:

```
#Variable1 *IS #ClassName  
#Variable2 *ISNOT *NULL
```

The *IS operator is used to test the class of the supplied variable, checking if the type of the variable or one of its ancestors matches the type identified by the class name.

The *ISNOT operator is used to test the class of the supplied variable, checking if the type of the variable or one of its ancestors does not match the type identified by the class name.

Both operators can also be used to test if the variable is a null reference.

Example

The following code fragment shows the use of the *IS and *ISNOT operators:

```
...  
DEFINE_COM CLASS(#DEPARTMENT) NAME(#DEPARTMENT)  
SET COM(#COM_OWNER) PSCENARIO('if cond(#DEPARTMENT *IsNot  
SET COM(#PASS) VALUE(FALSE)  
IF COND(#DEPARTMENT *IsNot *null)  
SET COM(#PASS) VALUE(TRUE)  
ENDIF  
  
SET COM(#COM_OWNER) PSCENARIO('if (#DEPARTMENT *IS #DEPTI  
(#ReferenceOne.Left <= 0))  
SET COM(#PASS) VALUE(FALSE)  
IF (#DEPARTMENT *Is #DEPARTMENT)  
SET COM(#PASS) VALUE(TRUE)  
ENDIF  
  
IF (((#DEPARTMENT *IsNot *null) *AndIf (#DEPARTMENT *Is #DEPTMI  
SET COM(#PASS) VALUE(TRUE)  
ENDIF  
...
```

↑ 8.28 Enhanced Expressions

8.28.7 *IsEqualTo and *IsOfType Operators

Just as with *IS and *ISNOT Operator, you can use the *IsEqualTo, *IsNotEqualTo, *IsOfType and *IsNotOfType operators to perform testing of component reference variables:

```
If (#COM_OWNER *IsEqualTo #Phbn_1.Parent)
Use Builtin(Ov_Message_box) With_Args("3. #COM_OWNER is equal to #P
Endif
If (#COM_OWNER *IsNotEqualTo #Phbn_1)
Use Builtin(Ov_Message_box) With_Args("4. #COM_OWNER is not equal to
Endif
If (#COM_OWNER *IsOfType #AADFORM22)
Use Builtin(Ov_Message_box) With_Args("5. #COM_OWNER is of type #A/
Endif
If (#COM_OWNER *IsNotOfType #PRIM_PHBN)
Use Builtin(Ov_Message_box) With_Args("6. #COM_OWNER is not of type
Endif
```

[↑ 8.28 Enhanced Expressions](#)

8.28.8 *AS Operator

You can perform type casting of component reference variables using the *AS operator. The following code fragment illustrates the syntax of the *AS operator:

```
If ((#Object1 *As #Prim_Form).Visible = True)
  Set #COM_OWNER Left((#Object1 *As #Prim_Form).Left)
EndIf
```

[↑ 8.28 Enhanced Expressions](#)

8.28.9 *ANDIF and *ORIF Logical Operators

RDMLX supports *AND and *OR logical operators. The following code fragment illustrates the syntax of these operators:

```
ConditionalExpression1 *OR ConditionalExpression2  
ConditionalExpression1 *AND ConditionalExpression2
```

Both of these operators fully evaluate the two conditional expressions coded either side of the operator before logically combining the resultant Boolean values into a logical result. This means that there is no way of stopping the second conditional expression based on the result of the first conditional expression.

Sometimes you may want to stop the second conditional expression based on the result of the first conditional expression. Referred to as short-circuiting, the following example illustrates the coding style that has previously been used logical operators.

```
If_Ref Com(#ReferenceOne) Is_Not(*null)  
  If Cond('#ReferenceOne.Left <= 0')  
    Set #ReferenceOne Left(100)  
  Endif  
Endif
```

So it has been necessary to write two conditional commands in order to first ensure that the variable #ReferenceOne was not null and therefore could be referenced. The second conditional command then checked the state of the component before executing the SET command.

Full RDMLX supports short circuiting using the *ANDIF and *ORIF operators. The *ANDIF operator will only execute the second conditional expression should the first conditional expression return a True result. The *ORIF operator will only execute the second conditional expression should the first conditional expression return a False result.

This means that the previous example can now be coded as:

```
If Cond( (#ReferenceOne *IsNot *Null) *AndIf (#ReferenceOne.Left <= 0))  
  Set #ReferenceOne Left(100)  
Endif
```

Example

The following code fragment shows the use of the *OrIf and *AndIf operators:

```
...
DEFINE_COM CLASS(#DEPARTMENT) NAME(#DEPARTMENT)
SET COM(#COM_OWNER) PSCENARIO('(#ReferenceOne *isnot *null) *and
(#ReferenceOne.Left <= 0))
IF COND((#DEPARTMENT *IsNot *null) *AndIf (#DEPARTMENT.Value = ADM
SET COM(#pass) VALUE(TRUE)
ENDIF
```

```
SET COM(#COM_OWNER) PSCENARIO('if (#DEPARTMENT.value = FIN) *and
IF ((#DEPARTMENT.value = ADM) *OrIf (#section.value = '05'))
SET COM(#pass) VALUE(TRUE)
ENDIF
```

```
SET COM(#COM_OWNER) PSCENARIO('if cond( (#COM_OWNER.Comp
IF ((#COM_OWNER.ComponentPatternName = QAP308) *AndIf (#com_self
SET COM(#pass) VALUE(TRUE)
ENDIF
```

...

[↑ 8.28 Enhanced Expressions](#)

9. Built-In Functions

LANSA is shipped with a number of Built-In Functions (BIFs) that perform common data processes.

Built-In Functions are invoked from RDML programs by the USE command. Before using any BIFs, please review: [USE](#) command.

Also see

[9.1 Built-In Function Rules](#)

For a full list of Built-In Function, refer to [Built-In Functions by Category Built In Functions \(BIFs\)](#) in the *Visual LANSAs Developer's Guide*.

[Create Your Own Built-In Functions](#) in the Application Design Guide.

9.1 Built-In Function Rules

Long Names

The majority of existing Built In Functions will accept only the Object Identifier when referring to LANSAs Objects. Built In Functions which support long names are documented accordingly.

BIF Argument & Return Value Types

An input argument or return value type can be of type A, N, L, U, w or X.

Type	Description
A	Alphanumeric - allows fields of type Alpha, String and Char.
N	Numeric - allows fields of type Packed, Signed, Float and Integer.
L	List
U	Unicode - allows fields of type Alpha, String, Char, NChar and NVarChar.
w	Any field type except Unicode may be supplied (excluding Lists). This is the same as X, but excluding Unicode.
X	Any field type may be supplied (excluding Lists).

All other field types like Date, DateTime and BLOB are classed as their own type and thus are not valid for either an argument/return type 'A', or type 'N'. To use these field types they must be coerced into the correct class using intrinsic functions. Refer to [Intrinsic Functions](#) for information and examples of using intrinsics.

User defined Built-In Functions require exact field type matches for all RDMLX field types. Therefore, if a String is used in the user defined Built-In Function declaration, then that is all that can be used when it is called. An Alpha field cannot be used. But if an argument is declared as Alpha, then String fields may be used. This anomaly exists due to support for backward compatibility. A user defined Built-In Function cannot declare an argument or return value of type X, U or w.

Fields v. Literals

When the length of an Argument is stated as being greater than 50, this is only

true for Fields. Literal values are restricted to a maximum length of 50.

FFC Warning instead of FFC Error for some arguments and return values in RDMLX

If an A, u, w or X (Any) argument or return value has a maximum length of 256, and a longer field is passed, this will be a FFC Warning (the BIF may not cope with fields over 256 bytes in length) instead of an error.

If an N argument or return value has a maximum length of 30 and maximum decimals of 9, and a longer field is passed, this will be a FFC Warning (the BIF may not cope with fields over 30,9) instead of an error.

Unlimited maximum length / maximum decimals in RDMLX

Some BIF arguments and return values will be defined with a new maximum length value of 2147483647, meaning **unlimited**. This will mean there is no need to check the min/max field length.

Some N or X arguments and return values will be defined with a new maximum decimals value of 32767, meaning unlimited. This will mean there is no need to check the min/max decimals length.

Fields of type Integer

Fields of type Integer have a size in bytes rather than a length, have no decimal places, and are accurate.

The following table provides the implied length for each of the possible byte lengths for an Integer. The implied length is equivalent to the actual length of a signed or packed field.

# Bytes	Max value (signed)	Max value (unsigned)	Max # digits (implied length)
1	127	255	3
2	32767	65535*	5
4	2147483647	4294967295*	10
8	9223372036854775807	18446744073709551615*	19 signed, 20 unsigned*

Fields of type Integer may only be used as numeric arguments or return values

under the following conditions:

- The minimum decimals for the argument or return value is 0.
- The minimum length for the argument or return value is less than or equal to the implied length of the Integer field. For example, if the minimum length for the argument is 4, an Integer of 1 byte may not be used (as it only has an implied length of 3).
- The maximum length for the argument or return value is 2147483647 OR the maximum length for the argument or return value is greater than or equal to the implied length of the Integer field. For example, if the maximum length for an argument is 4, an Integer of 2, 4, or 8 bytes may not be used (as they have implied lengths of 5 or higher).

Fields of type Float

Fields of type Float have a size in bytes rather than a length, can be assumed to contain decimal places although not of a fixed length, and are accurate to a certain number of digits.

The following table provides the accurate length for each of the possible byte lengths for a Float. The accurate length may be considered equivalent to the actual length of a signed or packed field. The table also notes the possible number of decimal places at runtime.

# Bytes	Accurate # digits (accurate length)	Possible decimal places
4	6	0 - 6
8	15	0 - 15

As the value for a field of type Float may have anywhere between 0 and 15 decimal places at execution time, it is generally not considered suitable as a numeric **argument** to a BIF as the actual number of decimal places cannot be predicted. An FFC warning will occur if a field of type Float is used for a numeric argument, unless the maximum length of the argument is 2147483647.

However, a field of type Float is suitable as a numeric **return value** under the following conditions:

- The maximum length is defined as 2147483647, meaning a number of any size is acceptable, OR The maximum decimals for the return value is 1 or higher, meaning a number with decimal places is normal and accepted.

- The minimum length and the minimum decimals for the return value are less than or equal to the accurate length of the Float field. For example, if the minimum length for the argument is 10, a Float of 4 bytes may not be used (as it is only accurate to 6 digits). Or, if the minimum decimals for the argument is 7 a Float of 4 bytes may not be used (as it is only accurate to 6 decimal places). This happens as part of normal numeric checking, given that a Float's length is adjusted as per the above table. If minimum length was 10 then a numeric of length 6 will cause an error regardless of the numeric type used. Likewise if the minimum decimal places is 7 then minimum length must be 7, or greater. Thus, a numeric of length 6 will again cause an error.
- Currently, the highest minimum decimals for any of LANSA's shipped BIF is 1. So, why the minimum decimals restriction? Because BIFs may be defined by customers (or new LANSA BIFs) that require a higher minimum # of decimal places.

Rules for Alphanumeric arguments and return values

Where an alphanumeric argument or return value is required, the following general rules apply in addition to the Built-In Function Argument & Return Value Types just listed.

- Fields of type **String** or **Char** may be used, as long as the field's length is within the range specified for the argument or return value. Note that if the maximum length is 2147483647, this means any length can be used.
- Fields of type **NChar** or **NVarChar** must be coerced to an Alphanumeric argument using the `asNativeString` intrinsic and have the same length restrictions as above. Refer to [asNativeString](#) for an example of using this Intrinsic.
- All other RDMLX Field types must be coerced to an Alphanumeric argument using the `asString` intrinsic and have the same length restrictions as above. Refer to the [Intrinsic Functions](#) for information and examples of using Ininsics.
- Fields of type **BLOB** and **CLOB** actually contain a filename (max length 256). To access the filename the syntax `#Myblob.Value` can be used as well as `#Myblob.asString`. (It is a developer decision as to whether or not the contents of the BLOB or CLOB filename are valid for the BIF argument or return value.)
- Fields of type **Date** (length always 10), **Time** (length always 8), and **DateTime** (length between 19 and 29) would require `asString` with length checks as specified. (It is a developer decision as to whether or not the

contents of the Date, Time, or DateTime are valid for the BIF argument or return value.)

Rules for Unicode arguments and return values

Argument and return types of X and U support Unicode

The rules for Unicode argument or return values are the same as for Alphanumeric, except :

- In RDML objects, a Unicode argument or return value is treated exactly the same as an Alphanumeric argument or return value.
- In RDMLX objects, fields of type NChar and NVarChar can be used directly and without data loss.
- If a Unicode argument uses a Unicode field, then a Unicode return value must use a Unicode field. This stops implicit data loss when Unicode is converted to native. Note that the reverse is not true. If a Unicode argument uses a native field, a Unicode return value may use either a Unicode field or a native field as there will be no implicit data loss.

Rules for List arguments and return values

The FFC does not check the aggregate byte length for working lists. If a specific aggregate byte length is required by the BIF you must ensure it is correct.

All Multilingual Built-In Functions

Non-DBCS SQL Server may corrupt DBCS data. DBCS SQL Server may corrupt all other language's text. To be sure of no corruption, only change text that is compatible with the database server's character set.

Note that the Visual LANSA integrated development environment performs database IO in a different way and so avoids corruption.

[↑ 9. Built-In Functions](#)

9.2 Development Environment only Built-In Functions

Some Built-In Functions are recommended for use with Development Environments only. These BIFs are identified by this link: **Development Environment only**.

These BIFs are restricted because they:

- require a LANSAs development license or hardware key (i.e. dongle) to use them. For example, if you intend to generate and compile LANSAs objects in a deployed environment, you will probably need some type of licensing on the system. Note that software key licenses do not support development environments-only BIFs.
- are designed purely to assist application developers and are not necessarily optimized for best performance in all situations.
- may use parts of the Visual LANSAs development environment. The Visual LANSAs development environment is rarely available in deployed Visual LANSAs environments.

The Visual LANSAs development environment is also 32-bit so 64-bit applications on Windows cannot use these BIFs.

- may access or update repository details.

In deployed systems this will cause complications if:

- the BIF assumes a development database is available on the system where the BIF is being executed.
- the appropriate repository details are not deployed with the application.
- the updated information is overwritten when repository details are later redeployed.

Note: When executing applications on Windows that use these development-only BIFs, the X_RUN argument LOCK=Y must be specified to ensure that object locks are released.

[↑ 9. Built-In Functions](#)

9.3 ACCESS_FILE

⇒ **Note:** Built-In Function Rules

Reads records from any file in the system including files not known to LANSA.

Warning:

When using this BIF ensure that the file and/or member exists. The job will fail if the file and/or member do not exist on the system.

The file must be opened before you can read records from it.

You cannot change a file/member once the file is open. To access a new file/member you must first close the open file/member.

If a library is not specified, the first file matching the requested file name in the library list will be used.

If a member is not specified the first member of the file will be used.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Action: OPEN - Open the file READ - Read a record CLOSE - Close the file	4	5		
2	A	Req	File name	1	10		

3	A	Opt	Library name Default *LIBL	1	10		
4	A	Opt	Member name	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK - Action completed ER - Error occurred EF - End of File	2	2		
2	A	Opt	Returned data block 1	1	256		
3	A	Opt	Returned data block 2	1	256		
4	A	Opt	Returned data block 3	1	256		
5	A	Opt	Returned data block 4	1	256		
6	A	Opt	Returned data block 5	1	256		
7	A	Opt	Returned data block 6	1	256		
8	A	Opt	Returned data block 7	1	256		
9	A	Opt	Returned data block 8	1	256		

If the records on the file are longer than 256 bytes, bytes 1-256 of the record will be returned in data block 1, bytes 257-512 of the record in data block 2, bytes 513-768 of the record in data block 3, etc.

Example

To read the first 10 records from a requested file and member.

```
DEFINE FIELD(#FILENM) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#LIBRARY) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#MEMBER) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#DATA1) TYPE(*CHAR) LENGTH(256)
*****
REQUEST FIELDS(#FILENM #LIBRARY #MEMBER)

USE BUILTIN(ACCESS_FILE) WITH_ARGS(OPEN #FILENM #LIBRA
#MEMBER) TO_GET(#RETCOD)

DUNTIL COND('#I *GE 10) *OR (#RETCOD *NE OK)')
USE BUILTIN(ACCESS_FILE) WITH_ARGS(READ #FILENM)
TO_GET(#RETCOD #DATA1)

* < process data1 >

ENDUNTIL

USE BUILTIN(ACCESS_FILE) WITH_ARGS(CLOSE
#FILENM....) TO_GET(#RETCOD)
```

9.4 ACCESS_RTE

⇒ **Note:** Built-In Function Rules.

Specifies or re-specifies the attributes of an access route between the definition of the file being edited and another file defined within the LANSAs system.

For details of what an access route is and how they are used by the LANSAs system refer to [Access Routes to Other Files](#) in the *LANSAs for i User Guide*.

After using this Built-In Function to define the basic access route attributes, repetitively use the ACCESS_RTE_KEY Built-In Function to specify or re-specify the route key field(s) or value(s).

An edit session must be commenced by using the START_FILE_EDIT Built-In Function prior to using ACCESS_RTE.

Allowable argument values and adopted default values are as shown in the [Detailed Access Route Maintenance](#) described in the *LANSAs for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSAs for i	YES	Do not use on an IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of access route First 3 letters of the name must be the same as the edit "source" nominated in the START_FILE_EDIT Built-In Function.	1	10		

2	A	Req	Description of access route	1	40		
3	A	Req	File to be accessed via route	1	10		
4	A	Req	Library in which file resides *FIRST and *DEFAULT are allowable. In Visual LANSA blanks or *LIBL are also valid for backward compatibility.	1	10		
5	N	Req	Maximum records expected Must be in range 1 - 9999.	1	4	0	0
6	A	Opt	Action to take if no records found via this route. Must be ABORT, IGNORE, N/AVAIL or DUMMY.	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = access route defined ER = error detected In case of "ER" return code error message(s) are issued automatically and the edit session continues. The access route that caused the error is ignored in this and all subsequent requests during the edit session.	2	2		

9.5 ACCESS_RTE_KEY

⇒ **Note:** Built-In Function Rules .

Specifies or re-specifies the name of a field or value that is to be used to access data via an access route previously defined via the ACCESS_RTE Built-In Function.

An edit session must be commenced by using the START_FILE_EDIT Built-In Function prior to using this Built-In Function.

Allowable argument values and adopted default values are described in [Detailed Access Route Maintenance](#) in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on an IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Access route name	1	10		
2	A	Req	Name of field from file being edited or a literal value that is to be used to form the key used to access data via the access route.	1	20		
3	N	Opt	Optional sequencing number. Used to sequence key fields. If not specified keys are sequenced in the same order as they are	1	5	0	0

		presented.				
--	--	------------	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = access key defined ER = error detected In the case of "ER" return code error message(s) are issued automatically and the edit session continues. The access route that caused the error is ignored in this and all subsequent requests during the edit session	2	2		

9.6 ADD_DD_VALUES

⇒ **Note:** Built-In Function Rules.

Adds either a new set of dropdown values or appends to an existing set of dropdown values.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES (Supported from V10.0)

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Dropdown name Must begin with DD	4	4		
2	A	Req	Separator Values: blank or *LOVAL means all one value	1	1		
3	A	Req	Dropdown value(s)	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code. Returned values possible are: OK: Value(s) added successfully ER: Error occurred	2	2		

Example

To set up the dropdown values for an order status field:

```
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
USE    BUILTIN(ADD_DD_VALUES) WITH_ARGS(DDST '/'
      'Raised/Open/Closed/Invoiced/History/Back Ordered')
      TO_GET(#RETCOD)
IF    COND('#RETCOD *NE OK')
* << error processing >>
ENDIF
USE    BUILTIN(ADD_DD_VALUES) WITH_ARGS(DDST '' 'Cancelled')
      TO_GET(#RETCOD)
IF    COND('#RETCOD *NE OK')
* << error processing >>
ENDIF
```

9.7 ALLOW_EXTRA_USER_KEY

⇒ **Note:** Built-In Function Rules.

Enables an "extra" user defined function key **above and beyond** any that are normally enabled by parameters on a DISPLAY, REQUEST or POP_UP command.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	A literal or variable that specifies or contains the number of the extra function key to be enabled. The value specified should be in the range 1 to 24 or it will be ignored.	1	2	0	0
2	A	Opt	Literal or variable that specifies or contains the description that should be associated with the function key when it is displayed in the function key area of a screen panel.	1	10		

Return Values

No return values.

Technical Notes

- Once an extra function key is enabled by ALLOW_EXTRA_USER_KEY, it

will affect the processing of all following DISPLAY, REQUEST or POP_UP commands within the current function.

- Each time ALLOW_EXTRA_USER_KEY is executed it adds another (or even the same) function key to a "stack" of extra keys that should be enabled by all following DISPLAY, REQUEST or POP_UP commands. This "stack" of extra enabled function keys can contain at most 24 entries. Attempting to enable more than 24 extra function keys (even if the same function key is enabled repeatedly) will cause an execution time application failure.
- All entries on the "stack" of extra function keys are removed by use of the DROP_EXTRA_USER_KEYS Built-In Function.
- Caution should be used to ensure that extra function keys enabled this way do not conflict with function keys automatically enabled by the normal parameters on a DISPLAY, REQUEST or POP_UP command. If such a conflict is allowed to occur then unpredictable or unexpected results may occur.
- The ability of this Built-In Function to dynamically enable and disable function keys, and to dynamically vary their associated descriptions, prevents the screen panel images used by the full function checker and screen painter from showing them in the image's function key area. Such function keys will only appear in the function key area of an executing application.

Examples

Enable function key 5 on all panels and pop-ups within a function:

```
FUNCTION OPTIONS( .....)  
USE    BUILTIN(ALLOW_EXTRA_USER_KEY) WITH_ARGS(5 'Refresh')
```

Enable function keys 18 and 19 on a particular panel and make sure that no other extra keys are accidentally enabled:

```
USE    BUILTIN(DROP_EXTRA_USER_KEYS)  
USE    BUILTIN(ALLOW_EXTRA_USER_KEY) WITH_ARGS(18 "'Hold'  
USE    BUILTIN(ALLOW_EXTRA_USER_KEY) WITH_ARGS(19 "'Save"
```

```
DISPLAY FIELDS(.....)
```

```
CASE   OF_FIELD(#IO$KEY)  
WHEN   VALUE_IS(= "18")  
      << hold processing >>
```

```
WHEN VALUE_IS(= "19")
  << save processing >>
ENDCASE
```

Enable extra function keys 14 to 21:

```
DEFINE FIELD(#I) TYPE(*DEC) LENGTH(2) DECIMALS(0)
USE BUILTIN(DROP_EXTRA_USER_KEYS)
BEGIN_LOOP FROM(14) TO(21) USING(#I)
USE BUILTIN(ALLOW_EXTRA_USER_KEY) WITH_ARGS(#I)
END_LOOP
```

9.8 BCONCAT

⇒ **Note:** Built-In Function Rules.

Concatenates up to five alphanumeric strings to form one string as a return value. Trailing blanks from each string are truncated and one blank is reinserted between each string during the concatenation operation.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	1st string to concatenate	1	Unlimited		
2	U	Req	2nd string to concatenate	1	Unlimited		
3	U	Opt	3rd string to concatenate	1	Unlimited		
4	U	Opt	4th string to concatenate	1	Unlimited		
5	U	Opt	5th string to concatenate	1	Unlimited		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	Concatenated result string	1	Unlimited		
2	N	Opt	Length of returned string	1	15	0	0

Example

Concatenate a first name and surname to get a print name field.

```
USE BUILTIN(BCONCAT) WITH_ARGS(#FNAME #SURNAME) TO_GET
```

9.9 BINTOHEX

⇒ **Note:** Built-In Function Rules.

Converts the contents of the source field from its binary format to an alphanumeric string consisting of two characters for each byte in the source.

For example, if Source contains AB, Return alphanumeric string will be C1C2 (IBM i) 4142 (Windows)

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	w	Req	Source: Visual LANSA Note: Source field type can be any RDMLX field type with unlimited length.	1	Unlimited		
2	N	Opt	Number of bytes to be converted. See Technical Note below.	1	11	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	X	Req	Returned string	2	Unlimited		

			Characters will be 0-9, A-F				
2	A	Opt	Return code. OK = action completed. ER = An error occurred.	2	2		

Technical Note

Second Argument	Built-In Function Behaviour
0 (Default)	<p>The value of the first argument (Source) is treated as a NULL terminated string:</p> <p>The first NULL byte in the source will be the terminator of the string.</p> <p>If there is no NULL byte in the source, the BIF will process the whole field.</p> <p>Any trailing BLANK, Carriage Return(CR) or Line Feed(LF) of the string being processed will be truncated.</p> <p>For example (in Windows): If the source value is 0xC1D4D840C1D4E2E8C4F0F220001F01 Then the actual value processed by this BIF will be: 0xC1D4D840C1D4E2E8C4F0F2 because the input value is only read until the first NULL(0x00) then the trailing BLANK (0x20) is truncated.</p>
Any negative value	The whole value of the source field will be processed. No truncation will happen.
Positive, not bigger than the current size of the first argument field	The BIF will process only the specified number of bytes from the source field. No truncation will happen to this portion of the source value.

Bigger than the current size of the first argument field	Return ER. No conversion will happen.
--	--

9.10 BUILD_WORK_OPTIONS

⇒ **Note:** Built-In Function Rules.

Dynamically converts a list of process and function names into a set of lists that are easy to use in "Work With" style RDML function drivers.

A list of process and function names are passed into this Built-In Function together with a "type" code. If type code is:

- O - this function acts on a single instance of an object in the work list.
- M - this function acts on multiple instances of objects in the work list.
- B - this function acts on multiple instances of objects in the work list (in a batch).

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	<p>Working list containing the function details.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 50 bytes.</p> <p>Each list entry should be formatted exactly as follows:</p> <p>Bytes 1-2: Option Number (packed format)</p> <p>Bytes 3-3: Function type (O, M or B)</p> <p>Bytes 4-13: Process Name</p> <p>Bytes: 14-20: Function Name</p>	50	50		

			<p>Bytes 21-50: Function Description</p> <p>This list is updated by the Built-In Function in accordance to the following rules:</p> <p>Any function that the user is not allowed to use is removed from the list.</p> <p>All "O" entries are removed and moved to the return lists.</p> <p>Any blank function descriptions are set to the correct current value (where it can be found).</p>				
2	N	Req	Entry length of the first list returned. This value must be in the range 40 to 80.	1	15	0	0

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	<p>Working list containing the Option Lines composed from all valid "O" entries of the 1st argument list.</p> <p>The calling RDML function must provide a working list of an aggregate entry length of exactly the length specified by the 2nd argument.</p>	1	80		
2	L	Req	<p>Working list containing the option numbers of those type "O" functions that were placed into the previous list as valid options in text format.</p> <p>The calling RDML function must provide a working list of an aggregate entry length of exactly 2 bytes.</p> <p>Each list entry should be formatted as follows:</p> <p>Bytes 1-2: Valid Option Number (packed</p>	1	2		

			format)				
--	--	--	---------	--	--	--	--

Examples

Imagine an input Definition List to this Built-In Function that contained entries like this:

Option	Type	Process	Function	Description
3	O	PROC01	FUNC01	Print Customer
4	O	PROC01	FUNC02	Change
5	O	PROC01	FUNC03	
0	M	PROC01	FUNC04	Print all Customers
0	M	PROC01	FUNC05	
0	B	PROC01	FUNC06	Print State Customer Sales
4	O	PROC01	FUNC07	Delete Customer

If this Built-In Function was executed, then it would return 3 lists that make "Work With" style functions easier to implement.

Returned Definition List

Option	Type	Process	Function	Description
0	M	PROC01	FUNC05	Send Outstanding FAXs
0	B	PROC01	FUNC06	Print State Customer Sales

Note

- All "O" entries have been removed.

- FUNC04 has been removed because the user is not authorized.
- The description of FUNC05 has been inserted.
- This list can be used to build a dynamic "menu" of functions to call (M) or submit (B).

Returned Textual List

Text (length 30, say)

3=Print Customer 4=Change

5=Send FAX to Customer

Note

- Two list entries are returned because all the text details would not fit into text "lines" of length 30.
- Options are "folded" so that they do not ever span an option text "line".
- The description of option 5 has been inserted.
- FUNC07 (option number 4) is not included because the user is not authorized to use it.
- This list can be used to dynamically build the options area on work with style screen panels.

Returned Valid Options List

Option

03

04

05

Note

- FUNC07 (option number 4) is not included because the user is not authorized

to use it.

- This list can be used to dynamically validate whether a user is authorized to an option number that they enter into a work with list.

9.11 CALL_SERVER_FUNCTION

⇒ **Note:** Built-In Function Rules.

Calls (executes) a LANSAs application on the nominated server and waits until it completes execution. The function must be a *DIRECT function.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	SSN of defined server.	1	10		
2	A	Req	Name of function to be called.	1	7		
3	A	Opt	Pass Exchange List Y= Pass exchange list. other = do not pass exchange list. The default is N.	1	1		
4	A	Opt	Return Exchange List Y= Return exchange list. other = do not return exchange list. The default is N.	1	1		
5 - 14	L	Opt	Working Lists 1 through 10 to be passed to the function on the server.				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Call Completed ER - Error during call and error message(s) issued.	2	2		

Technical Notes

- The server function that is called is strictly a "batch" job on the server. It cannot execute DISPLAY/REQUEST/POP_UP commands in any form (e.g., via Windows) because it is not logically connected to any form of user interface. Likewise it cannot successfully use other IBM i products that need to be logically connected to a user interface (e.g., STRSEU, WAF/400).

This point needs to be understood very clearly because it may influence application design. Client functions "talk" to the user via DISPLAY/REQUEST/POP_UP commands. Server functions are actually executing on the server and thus have no direct or logical way by which they can "talk" to the user.

A server function is typically a "subroutine" of a client function that just happens to be executed on another platform.

- Fields exchanged or passed in working lists are automatically converted from ASCII to EBCDIC (and back again). The conversions are invisible to client and server functions. It is just as if the function was being called on the same machine via the RDML CALL command.
- The same rules that apply to the RDML CALL command for working list passing / receiving apply to this Built-In Function.

For example: All working lists must have identical definitions in the caller

and receiver at all times. If the list definition changes, both must be recompiled.

- Fields can be exchanged to, and back from, the server function.
- Working lists can be passed to, and received back from, the server function.
- Alphanumeric fields with DBCS attributes are translated to/from ASCII and EBCDIC DBCS. This type of translation only occurs when the field has DBCS attributes (e.g., J, E, O) and the server has been connected with the "DBCS Capable" option set to Y.
- When the working list details only need to be passed to the server function, make the server function clear the working list before it completes. This saves having to send the list back to the client again and reduces communications traffic.
- When a working list only needs to be returned by the server function, clear it before calling the server function. This saves having to send the list details to the server and thus reduces communications traffic.
It may also prevent accidental "overflowing" when the server function assumes that it is receiving an empty or cleared list. See the following points for more details about "overflowing".
- Message information routed from the server machine (in any form) arrives in a text format. It is displayed and accessible to RDML functions in the normal manner (e.g., GET_MESSAGE) as pure text. The message identifier and message file name details are not available for messages that have been routed from a server. You should not design client applications that rely on reading specific message identifiers from the applications message queue.

Portability Considerations Servers defined with DEFINE_OS_400_SERVER:

The aggregate byte length of a working list passed to a server cannot exceed 32,000 bytes. The aggregate byte length is the entry byte length multiplied by the current number of entries. As 1 to 10 working lists can be passed to the function on the server, the total number of bytes that can be passed to the Server is 320,000, that is, 10 working lists of 32,000 bytes each.

This Built-In Function will cause a fatal error message if a client function passes a list that is too large. However, the server function is a different matter. The working list it

receives as a parameter is in memory allocated by its caller (i.e. the IBM i based server controller). If it attempts to add too many entries to the working list it may "zap" the server controller and cause an application failure.

Please do not ignore this warning. Server (i.e. IBM i based) functions that receive working lists from Client (e.g.

Windows) functions via this Built-In Function must take great care not to "overflow" the working list(s) that they are passed.

If an unexpected failure of CALL_SERVER_FUNCTION occurs, and working lists are involved, then look to this point as the first possible cause the server function may be overflowing the working list(s).

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application.

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
endif
```

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

9.12 CENTRE

⇒ **Note:** Built-In Function Rules.

Centers argument string into return string.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

The following table shows the arguments used in this function.

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be centered	1	256		
2	A	Opt	Remove imbedded blanks flag Values: Y = remove N = do not remove Default: N	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return centered string	1	256		

9.13 CHANGE_IBMI_SIGNON

⇒ **Note:** Built-In Function Rules.

Changes the password of the user profile on the IBM i server.

For use with

LANSA for i	YES	Only available with RDMLX
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	IBM i Server Name or IP Address	1	256		
2	A	Req	SSL Required. Y - Use SSL to communicate with the IBM i server other - do not use SSL	1	1		
3	A	Req	User name (signon) to be changed	1	10		
4	A	Req	Password for the User Name to be changed.	1	128		
5	A	Req	New Password for the User Name.	1	128		
6	A	Req	Encrypt Password. Y - Encrypt password other - do not encrypt password	1	1		
7	N	Opt	Server Mapper Port. Defaults to 449 if not specified or passed as 0.	1	5	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Password changed OK CE - Communications error LE - Local Encryption error NR - User name does not exist PE - Password Expired SE - Server error WP - Wrong Password UD - User name is disabled LP - Password is too long NE - New Password Error	2	2		

Technical Notes

Because of the way that the IBM i operating system handles user names and short passwords (Password level 0 or 1) with the US English (CCSID 037) characters '@', '#' and '\$', this facility will only work with such user names and short passwords if the IBM i is operating in US English (CCSID 037).

The current implementation of SSL used for this facility ensures that encryption is negotiated and used for communication between the client and the IBM i server. It does not verify that the IBM i server is that specified on the security certificate that has been downloaded.

The interplay between SSL Required and Encrypt Password is interesting. If SSL is available and SSL Required is Y, then strictly speaking password encryption is not needed because the entire communication stream is encrypted, so Encrypt Password could be specified as N. If SSL Required is N, then we recommend that Encrypt Password be specified as Y.

The reasons for Return Code CE -Communications error can include:

- a misspelling in the IBM i Server name;
- the IBM i Server name not being locatable by your DNS;
- a firewall between the local computer and the IBM i server;
- the IBM i server being offline;
- TCP/IP not being started on the IBM i server;
- TCP/IP host servers not being started on the IBM i server;
- SSL Required Y and the SSL TCP/IP host servers not being started on the IBM i server;
- SSL not required and the non-SSL TSP/IP host servers not being started on the IBM i server.

If Return Codes SE - Server Error or NE - New Password Error is returned, a review of the joblog for the QZSOSIGN job on the IBM i server should show the detailed reason.

9.14 CHECK_AUTHORITY

⇒ **Note:** Built-In Function Rules.

Checks whether a user has a certain authority to an object.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	Object Types P# and AT have no meaning in the context of this platform. If either of these object types are passed to this Built-In Function, an error is returned.
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object name	1	10		
2	A	Req	Object name extension Values: - blanks - literal - *LIBL	1	10		
3	A	Req	Object type Values: DF - Field AT - Application template* PF - Function PD - Process	2	2		

			FD - File P# - Partition* SV - System variable MT - Multilingual string. *Note: Object Types P# and AT have no meaning in the context of Visual LANSA.				
4	A	Req	Access required to object Values: Operational UD - Use definition MD - Modify definition DD - Delete definition Data DS - Display AD- Add CH - Change DL - Delete	2	2		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Access granted	1	1		

This function can be used to determine whether a user can update a record in a file OR use a field definition.

```

DEFINE FIELD(#OBJEXT) TYPE(*CHAR) LENGTH(10) DEFAULT(' '*LI
* < OR >
DEFINE FIELD(#OBJEXT) TYPE(*CHAR) LENGTH(10) DEFAULT(*BL
USE BUILTIN(CHECK_AUTHORITY) WITH_ARGS (#FILENAME #OI
IF COND('#OKAY *EQ Y')
UPDATE FILE(#FILENAME)
ELSE

```

```

MESSAGE MSGTXT('Not authorized to update file')
ENDIF
DEFINE FIELD(#OBJEXT) TYPE(*CHAR) LENGTH(10) DEFAULT(*BL.
USE    BUILTIN(CHECK_AUTHORITY) WITH_ARGS #FLDNAME #OB.
IF    COND('#OKAY *EQ Y')
USE    BUILTIN(GET_FIELD) WITH_ARGS(#FIELDNAME) TO_GET(#R
ENDIF

```

When the value *CHECK_AUTH_DYNLIBL is not specified in data area DC@OSVEROP, when checking authorities on files (FD) and no library or *LIBL is specified, it is assumed the file is in the library list that was present at time of entry to LANSAs. Library list cannot be changed dynamically. If the library list was changed during the LANSAs session LANSAs must be left and re-entered for the new library list to be recognized.

Tips

When the value *CHECK_AUTH_DYNLIBL is specified in data area DC@OSVEROP, when checking authorities on files (FD) and no library or LIBL is specified, the library list is retrieved each time dynamically to determine which library the CHECK_AUTHORITY is to actually use. Use of this setting may have performance implications.

When checking authority on files use the system variable *PARTDTALIB for the object extension value.

Note: When checking FUNCTION authorization (type PF) both the PROCESS and the FUNCTION must be specified as arguments. That is, the PROCESS is the object name and the FUNCTION is the object extension.

```

USE    BUILTIN(CHECK_AUTHORITY)
      WITH_ARGS (#PROCESS CONTROL PF UD ) TO_GET(#OKAY)
IF    COND('#OKAY *EQ Y')
CALL   PROCESS(#PROCESS) FUNCTION(CONTROL)
ELSE
MESSAGE MSGTXT('Not authorized to function CONTROL')
ENDIF

```

9.15 CHECK_IBMI_SIGNON

⇒ **Note:** Built-In Function Rules.

Checks the status of the user profile on the IBM i server.

For use with

LANSA for i	YES	Only available with RDMLX
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	IBM i Server Name or IP Address	1	256		
2	A	Req	SSL Required. Y - Use SSL to communicate with the IBM i server other - do not use SSL	1	1		
3	A	Req	User name (signon) to be checked	1	10		
4	A	Req	Password for the User Name to be checked.	1	128		
5	A	Req	Encrypt Password. Y - Encrypt password other - do not encrypt password	1	1		
6	N	Opt	Server Mapper Port. Defaults to 449 if not specified or passed as 0.	1	5	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Signon is OK CE - Communications error LE - Local Encryption error NR - User name does not exist SE - Server error WP - Wrong Password UD - User name is disabled LP - Password is too long	2	2		
2	A	Req	Date Password Expires. Only valid if the Return Code is OK. Returned as '99991231' if the password never expires.	8	8		

Technical Notes

Because of the way that the IBM i operating system handles user names and short passwords (Password level 0 or 1) with the US English (CCSID 037) characters '@', '#' and '\$', this facility will only work with such user names and short passwords if the IBM i is operating in US English (CCSID 037).

The current implementation of SSL used for this facility ensures that encryption is negotiated and used for communication between the client and the IBM i server. It does not verify that the IBM i server is that specified on the security certificate that has been downloaded.

The interplay between SSL Required and Encrypt Password is interesting. If SSL is available and SSL Required is Y, then strictly speaking password encryption is not needed because the entire communication stream is encrypted, so Encrypt Password could be specified as N. If SSL Required is N, then we recommend that Encrypt Password be specified as Y.

The reasons for Return Code CE -Communications error can include:

- a misspelling in the IBM i Server name;
- the IBM i Server name not being locatable by your DNS;
- a firewall between the local computer and the IBM i server;
- the IBM i server being offline;
- TCP/IP not being started on the IBM i server;
- TCP/IP host servers not being started on the IBM i server;
- SSL Required Y and the SSL TCP/IP host servers not being started on the IBM i server;
- SSL not required and the non-SSL TSP/IP host servers not being started on the IBM i server.

If Return Code SE - Server Error is returned, a review of the joblog for the QZSOSIGN job on the IBM i server should show the detailed reason.

9.16 CHECKNUMERIC

⇒ **Note:** Built-In Function Rules.

Checks a string only contains allowable values and converts the digital and decimal portions into numeric variables.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be converted	1	256		
2	N	Opt	Max. no. of integers allowed Range: 0 - 63 Default: 15	1	3	0	0
3	N	Opt	Max. no. of decimals allowed Range: 0 - 63 Default: 9	1	3	0	0
4	A	Opt	List of allowable characters to be ignored e.g. \$, %, C, R	1	50		

Return Values

No	Type	Req/ Opt	Description	Min	Max Len	Min	Max Dec
----	------	-------------	-------------	-----	---------	-----	---------

		Opt		Len		Dec	
1	N	Req	Return integer portion	1	Unlimited	0	0
2	N	Opt	Return decimal portion	1	Unlimited	1	Unlimited
3	A	Opt	Return okay code (Y/N)	1	1		
4	A	Opt	Return sign of the number (+ or -)	1	1		
5	N	Opt	Return number of integers	1	3	0	0
6	N	Opt	Return number of decimals	1	3	0	0

Example

To get a packed decimal 9,2 result field #P92 from an alphanumeric field #A using 2 intermediate work fields called #P90 and #DEC.

```
USE      BUILTIN(CHECKNUMERIC) WITH_ARGS(#A 7 2) TO_GET(#P92)
CHANGE  FIELD(#P92) TO('#P90 + #DEC')
```

9.17 CHECKSTRING

⇒ **Note:** Built-In Function Rules.

Checks a string contains only allowable characters.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be checked	1	Unlimited		
2	A	Req	List of allowable characters	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return okay code (Y/N) Y = only contains specified characters N = contains other characters	1	1		

Technical Notes

Alpha fields always contain trailing blanks up to the length of the field. These trailing blanks need to be considered if you use Alpha fields when using this BIF. If trailing blanks are permitted in the String to be checked, then a blank must be provided in the list of allowable characters.

A literal may be used for the List of allowable characters.

If an Alpha field is used for the List of allowable characters, it may contain trailing blanks. For example

```
USE BUILTIN(CHECKSTRING) WITH_ARGS(#STRING #ALLOW)
  TO_GET(#YN)
```

If #ALLOW is Alpha(2) with a value of 'A ', then characters A and blank are checked for in #STRING.

If #ALLOW contains 'AB', only characters A and B are checked for in #STRING. Any trailing blanks in #STRING will result in a return code of N (contains other characters).

Reminder - in RDMLX you may:

- Use the intrinsic .Trim on the Built-In Function arguments to trim trailing blanks from field values.
- Replace the CHECKSTRING Built-In Function with the ContainsOnly intrinsic. This intrinsic treats trailing blanks in Alpha fields as insignificant and handles DBCS characters.
- Use fields of type String

9.18 CLR_MESSAGES

⇒ **Note:** Built-In Function Rules.

Clears all messages from the RDML program queue function.

Messages on the program queue of an RDML function are normally displayed on line 22/24 of the next screen presented to the user and then automatically cleared / removed.

Messages may have been placed on the program message queue by operating system commands, Built-In Functions, invalid I/O requests and/or RDML commands such as MESSAGE, VALUECHECK, etc.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Clear Event Log Messages (Y/N) This option only affects Visual LANSA. Default is Y to clear the Event Log messages.	1	1		

Return Values

No values are returned by this Built-In Function.

9.19 COMPARE_FILE_DEF

⇒ **Note:** Built-In Function Rules.

Compares two CTD files and returns a flag to indicate if the objects are different.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	The LANSA File Name	1	10		
2	A	Req	Path for the current CTD	1	256		
3	A	Req	Path for the new CTD	1	256		
4	A	Opt	Library Name	1	10		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code NC = No differences found CH = Differences were found in	2	2		

		the definitions				
		ER = An error occurred				

9.20 COMPILE_PROCESS

⇒ **Note:** Built-In Function Rules.

Compiles a process and all selected functions.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition. LANSA for i submits a job to batch as a separate task.
Visual LANSA for Windows	YES	Visual LANSA initiates the compile process and does not return control until the compile is complete.
Visual LANSA for Linux	NO	

Arguments for Visual LANSA

No	Type	Req/Opt	Visual LANSA Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Process name	1	10		
2	L	Req	Working list to contain function names. The calling RDML function must provide a working list with an aggregate entry length of exactly 7 bytes. If you do not wish to specify any functions for compilation then you must pass an empty working list.	1	7		
3	A	Opt	Name of job Ignored	1	10		
4	A	Opt	Name of job description	1	21		

			Ignored				
5	A	Opt	Name of job queue Ignored.	1	21		
6	A	Opt	Name of output queue Ignored.	1	21		
7	A	Opt	Force compile of the process. This option forces the process to be rebuilt. If this option is NO, the process may still be compiled if changes have been made since it was last compiled. Functions are not affected. Functions are always compiled. Default NO.	2	3		
8	A	Opt	Produce RDML source listing? Ignored.	2	3		
9	A	Opt	Produce RPG & DDS source listings? This option is treated as the <i>Keep Source</i> option in LANSA/PC.	2	3		
10	A	Opt	Optimize compiled program(s)? Ignored.	2	3		
11	A	Opt	Ignore decimal data errors in program(s)? Ignored.	2	3		
12	A	Opt	Allow debug / Remove Program observability? Only the first character is checked. If this is Y then debug is enabled. Default Y	6	6		
13	A	Opt	Dump code generator work areas? Ignored.	2	3		
14	A	Opt	Produce Documentor details? Ignored.	2	2		
15	A	Opt	Generate HTML pages?	2	3		

			<p>YES = Generate HTML pages NO = Do not generate HTML pages Default : YES Note: This argument is only applicable to Web enabled processes.</p>				
16	A	Opt	<p>Generate HTML editor extension details? This option is treated the same as "Validate Numeric Values" YES = Generate details to support HTML editor extension. NO = Do not generate details. Note: This argument is only applicable to web enabled processes. If this option is YES, the Generate HTML Pages option must also be YES.</p>	2	3		
17	A	Opt	<p>Generate XML? YES = Generate XML NO = Do not Generate XML Default: YES Note: This argument is only applicable to XML enabled processes</p>	2	3		

Arguments for LANSA for i

No	Type	Req/ Opt	LANSA for i Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Process name	1	10		
2	L	Req	Working list to contain function names. The calling RDML function must provide a working list with an aggregate entry length of exactly 7 bytes.	1	7		

			<p>If you do not wish to specify any functions for compilation then you must pass an empty working list.</p> <p>Each returned list entry is formatted as follows:</p> <p>Bytes 1-7: Function name</p>				
3	A	Opt	<p>Name of batch job</p> <p>Default: Process name</p>	1	10		
4	A	Opt	<p>Name of job description</p> <p>Default: the job description from the requesting job's attributes.</p>	1	21		
5	A	Opt	<p>Name of job queue</p> <p>Default: the job queue from the requesting job's attributes.</p>	1	21		
6	A	Opt	<p>Name of output queue</p> <p>Default: the output queue from the requesting job's attributes.</p>	1	21		
7	A	Opt	<p>Compile process as well as functions?</p> <p>YES = compile process</p> <p>NO = do not compile process</p> <p>Default: the "compile process default" value at position 461 in the system definition data area DC@A01. See</p> <p>Note</p>	2	3		
8	A	Opt	<p>Produce RDML source listing?</p> <p>YES = produce RDML listing</p> <p>NO = do not produce listing</p> <p>Default: the "source listing default" value at position 146 in the system definition data area DC@A01.</p>	2	3		
9	A	Opt	<p>Produce RPG & DDS source listings?</p> <p>YES = produce RPG & DDS listings</p>	2	3		

			<p>NO = do not produce listings</p> <p>Default: the "source listing default" value at position 146 in the system definition data area DC@A01. See</p> <p>Note</p>				
10	A	Opt	<p>Optimize compiled program(s)?</p> <p>YES = optimize program(s)</p> <p>NO = do not optimize</p> <p>Default: the "optimize compile default" value at position 147 in the system definition data area DC@A01. See</p> <p>Note</p>	2	3		
11	A	Opt	<p>Ignore decimal data errors in program(s)?</p> <p>YES = ignore decimal data errors</p> <p>NO = do not ignore errors</p> <p>Default: the "decimal data error default" value at position 148 in the system definition data area DC@A01. See</p> <p>Note</p>	2	3		
12	A	Opt	<p>Allow debug / Remove Program observability?</p> <p>YESYES = Allow program(s) to be used in debug and do not remove observability.</p> <p>NO NO = Do not allow debug and remove program observability</p> <p>NO YES = Do not allow debug but do not remove the programs observability.</p> <p>Default: the "enable debug default" value at position 400 in the system definition data area DC@A01.</p> <p>Warning: Do not specify YESNO for this parameter. The DEBUG facility cannot work if a program is not observable.</p>	6	6		

13	A	Opt	<p>Dump code generator work areas?</p> <p>YES = Dump work areas</p> <p>NO = Do not dump work areas</p> <p>Default: YES</p>	2	3		
14	A	Opt	<p>Produce Documentor details?</p> <p>YES = Produce Documentor details</p> <p>NO = Do not produce Documentor details</p> <p>Default: YES if Documentor is enabled at the partition level, otherwise NO.</p>	2	2		
15	A	Opt	<p>Generate HTML Pages?</p> <p>YES = Generate HTML pages</p> <p>NO = Do not generate HTML pages</p> <p>Default : YES</p> <p>Note: This argument is only applicable to Web enabled processes.</p>	2	3		
16	A	Opt	<p>Validate numerics</p> <p>YES = Generate details to support HTML editor extension.</p> <p>NO = Do not generate details.</p> <p>Note: This argument is only applicable to web enabled processes.</p> <p>If this option is YES, the Generate HTML Pages option must also be YES.</p>	2	3		
17	A	Opt	<p>Generate XML?</p> <p>YES = Generate XML</p> <p>NO = Do not Generate XML</p> <p>Default: YES</p> <p>Note: This argument is only applicable to XML enabled processes</p>	2	3		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = successful submission. ER = argument details are invalid or an authority problem has occurred. In this case, return code error message(s) are issued automatically.	2	2		

Notes for LANSAs for i Arguments

No	
9	Produce RPG & DDS source listings? Default: the "source listing default" value at position 146 in the system definition data area DC@A01. Review this default setting via the <i>Compile process</i> option in Work with Compile and Edit Settings in the Review Systems setting facility.
10	Optimize compiled program(s)? Default: the "optimize compile default" value at position 147 in the system definition data area DC@A01. Review this default setting via the <i>Produce source listing</i> option in Work with Compile and Edit Settings in the Review Systems setting facility.
11	Ignore decimal data errors in program(s)? Default: the "decimal data error default" value at position 148 in the system definition data area DC@A01. Review this default setting via the <i>Ignore Decimal data errors in RPG</i> option in Work with Compile and Edit Settings in the Review Systems setting facility.

Example

A user wants to control the compilation of processes and functions using their own version of the "Compile / Re-Compile a Process" facility.

```
***** Define arguments and lists
DEFINE FIELD(#PROCES) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#FUNCTN) TYPE(*CHAR) LENGTH(7)
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
DEF_LIST NAME(#WKFUNL) FIELDS((#FUNCTN)) TYPE(*WORKING
DEF_LIST NAME(#BWFUNL) FIELDS((#FUNCTN))
***** Clear working and browse lists
BEGIN_LOOP
CLR_LIST NAMED(#WKFUNL)
INZ_LIST NAMED(#BWFUNL) NUM_ENTRIES(10) WITH_MODE(*CH/
***** Request Process and Functions
REQUEST FIELDS(#PROCES) BROWSELIST(#BWFUNL)
***** Move Functions from the browselist to the working list
SELECTLIST NAMED(#BWFUNL)
ADD_ENTRY TO_LIST(#WKFUNL)
ENDSELECT
***** Execute built-in-function - COMPILE_PROCESS
USE BUILTIN(COMPILE_PROCESS) WITH_ARGS(#PROCES #WKFUNL)
***** Check if submission was successful
IF COND(#RETCOD *EQ "OK")
MESSAGE MSGTXT('Compile Process submitted successfully')
CHANGE FIELD(#PROCES) TO(*BLANK)
ELSE
MESSAGE MSGTXT('Compile Process submit failed with errors, refer to ac
ENDIF
END_LOOP
```


4	A	Opt	Compile for debug (YES/NO). Default: NO	2	3		
5	A	Opt	Web services to compile (A/W/N) A - All Web Routines W - New Web Routines only N - None. Default: N	1	1		
6	L	Opt	Working List Name. The working list must have an aggregate length of 21 bytes. Each list entry should be formatted as follows: Technology Services identifier. A(21). This should be specified in the following format <Provider>:<Technology Service Name>. For example, LANSA:XHTML	1	21		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code. The component may fail if it doesn't pass the build process or is locked. OK = successful submission of compile. NR = No compilable components found in list. ER = argument details are invalid or an authority problem has occurred. In this case, return code error message(s) are issued automatically.	2	2		

Example

A user wants to control the compilation of components using their own version of the "Compile / Re-Compile a Component" facility.

```
***** Define arguments and lists
DEFINE  FIELD(#COMPNAME) TYPE(*CHAR) LENGTH(9)
DEFINE  FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
DEF_LIST  NAME(#WKCOMP) FIELDS(#COMPNAME) TYPE(*WORKI
DEF_LIST  NAME(#BWCOMP) FIELDS(#COMPNAME)
***** Clear working and browse lists
BEGIN_LOOP
CLR_LIST  NAMED(#WKCOMP)
INZ_LIST  NAMED(#BWCOMP) NUM_ENTRYS(2) WITH_MODE(*CHA
***** Request component names
REQUEST  BROWSELIST(#BWCOMP)
***** Move components from the browselist to the working list
SELECTLIST NAMED(#BWCOMP)
ADD_ENTRY TO_LIST(#WKCOMP)
ENDSELECT
***** Execute built-in-function - COMPILE_COMPONENT
USE      BUILTIN(COMPILE_COMPONENT) WITH_ARGS(#WKCOMP)
***** Check if submission was successful
IF      COND(#RETCOD *EQ "OK")
MESSAGE MSGTXT('Compile Component submitted successfully')
ELSE
MESSAGE MSGTXT('Compile Component submit failed with errors, refer t
ENDIF
END_LOOP
```

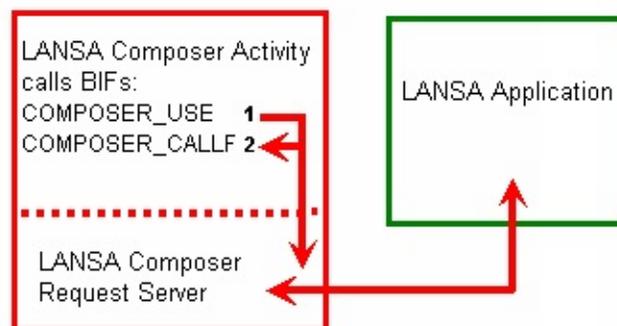
9.22 COMPOSER_CALLF

⇒ **Note:** Built-In Function Rules.

This Built-In Function can only be used with LANSAs Composer greater than V3.0.

This Built-In Function assumes that the source system (the system in which the Built-In Function executes) is a LANSAs Composer system, greater than Version 3, and contains the LANSAs Composer Request Server software. **This Built-In Function will fail if this is not the case.**

COMPOSER_CALLF calls a named LANSAs function, in the LANSAs or LANSAs Composer system identified by the server symbolic name argument, through the LANSAs Composer Request Server.



It can pass and receive up to seven values via the LANSAs exchange list. All exchange variables are passed as A(256) using exchange variable names EXCH01 - EXCH07.

This Built-In Function is intended for use in custom activity processors or other user-defined plug-in components of LANSAs Composer to invoke processing logic contained in another LANSAs application that may be installed in a different LANSAs system and/or partition.

You must execute the [9.24 COMPOSER_USE](#) Built-In Function to define the server connection details and a symbolic name representing them before executing this Built-In Function.

Further important information about this Built-In Function is provided later in these notes in:

Exchange Variables

LANSA programming considerations for the called function

Further considerations for functions calls executed through the LANSAComposer Request Server

You should also refer to LANSAComposer documentation of the CALL_FUNCTION activity and of the LANSAComposer Request Server for further information and considerations.

For use with

LANSA for i	YES
Visual LANSAComposer for Windows	YES
Visual LANSAComposer for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	LANSA Composer server symbolic name. Specifies a symbolic name used to identify the connection details for the required LANSAComposer server system. The name must have previously been specified in the COMPOSER_USE Built-In Function in the current session.	1	10		
2	A	Req	Name of the LANSAComposer process containing the function to be called. IBM i : if *DIRECT is specified, the function must be defined with FUNCTION OPTIONS(*DIRECT). Windows servers : the process name (not *DIRECT) must be specified.	1	10		

3	A	Req	Name of the LANSAs function to be called. Compulsory.	1	7		
4	N	Opt	Number of exchange variables used. If not specified, a default of 0 (zero) is assumed.	1	5	0	0
5	A	Opt	Value of exchange variable EXCH01 (see Exchange Variables).	1	256		
6	A	Opt	Value of exchange variable EXCH02 (see Exchange Variables).	1	256		
7	A	Opt	Value of exchange variable EXCH03 (see Exchange Variables).	1	256		
8	A	Opt	Value of exchange variable EXCH04 (see Exchange Variables).	1	256		
9	A	Opt	Value of exchange variable EXCH05 (see Exchange Variables).	1	256		
10	A	Opt	Value of exchange variable EXCH06 (see Exchange Variables).	1	256		
11	A	Opt	Value of exchange variable EXCH07 (see Exchange Variables).	1	256		
12	A	Opt	Synchronous call? Specifies whether the Built-In Function waits for the function call to complete. Default is 'Y', to wait. If any other value is specified, the Built-In Function posts the function call request and ends immediately. Note that this Built-In Function can only receive values returned from the called function (in the EXCH01 ... EXCH07 exchange variables) if this parameter is 'Y'.	1	1		
13	N	Opt	Synchronous time-out (seconds). The number of seconds the Built-In Function waits for a synchronous call to complete. If	1	5	0	0

4	A	Opt	Value of exchange variable EXCH03 (see Exchange Variables).	1	256		
5	A	Opt	Value of exchange variable EXCH04 (see Exchange Variables).	1	256		
6	A	Opt	Value of exchange variable EXCH05 (see Exchange Variables).	1	256		
7	A	Opt	Value of exchange variable EXCH06 (see Exchange Variables).	1	256		
8	A	Opt	Value of exchange variable EXCH07 (see Exchange Variables).	1	256		

Examples

This example uses a previously defined connection to a LANSAsystem, with the symbolic name COMPOSER (see COMPOSER_USE), to do a simple function call. The process name is specified in the variable #PROCESS and the function name is specified in the variable #FUNCTION.

```
use builtin(COMPOSER_CALLF) with_args('COMPOSER' #PROCESS
#FUNCTION)
```

This example connects to the same LANSAsystem as the above to call a function. The process name is specified in the variable #PROCESS and the function in the variable #FUNCTION. Two parameter values are passed: 'VALUE 1' and 'VALUE 2'. To receive these values through the exchange list, the called function must have fields EXCH01 and EXCH02 defined. In this case, the activity will not wait for the function to complete before proceeding as the call is asynchronous.

```
use builtin(COMPOSER_CALLF) with_args('COMPOSER' #PROCESS
#FUNCTION 2 'VALUE 1' 'VALUE 2' *Default *Default *Default *Default
*Default N) to_get(#RESULT)
```

This example connects to the same LANSAsystem as the above to call a function. The process name is specified in the variable #PROCESS and the function in the variable #FUNCTION. Two parameter values are passed and

three are returned. In order to receive these variables via the exchange list, the called function must have the fields EXCH01 and EXCH02 defined. In addition, to return the variables the field EXCH03 must also be defined and the fields must be exchanged.

```
use builtin(COMPOSER_CALLF) with_args('COMPOSER' #PROCESS
#FUNCTION 2 'VALUE 1' 'VALUE 2') to_get(#RESULT #VAR1 #VAR2
#VAR3)
```

Exchange Variables

The Built-In Function arguments and return values can be used to pass and receive up to seven values to/from the called function via the LANSAs exchange list. The parameters are placed on and received from the exchange list as character variables of length 256 using the variable names EXCH01 ... EXCH07.

The called function must also use the variables names EXCH01 ... EXCH07 in order to receive the exchange values. If the called function needs to return values via these variables, it must execute the EXCHANGE command at the appropriate point.

The Built-In Function will place on and receive from the exchange list the number of parameters (up to seven) specified in the fourth argument. If used, they must be specified contiguously - for example, if you specify the value three, the Built-In Function will exchange the variables EXCH01, EXCH02 and EXCH03 and the values for the remaining exchange variable arguments will be ignored.

Note that the Built-In Function can only receive values returned from the called function when executed synchronously.

Refer to the description of the [EXCHANGE](#) command for further information on exchanging information via the exchange list.

LANSAs programming considerations for the called function

- If *DIRECT is specified or assumed for the PROCESS parameter, the function must be defined with FUNCTION OPTIONS(*DIRECT)
- Called functions may be RDML or they may be fully RDMLX enabled.
- The function must define fields EXCH01 ... EXCH07 in order to receive values (via the exchange list) that are specified in the corresponding Built-In Function arguments.

- The function must use the EXCHANGE command with fields EXCH01 ... EXCH07 in order to return values (via the exchange list) to populate the corresponding Built-In Function return values.
- On IBM i servers only, position 487 of LANSAs data area DC@A01 in the LANSAs system containing the function to be called must be set to 'Y' before compiling or executing the function. If this condition is not met, the called function will not correctly receive or return the EXCH01 ... EXCH07 variable values.

Depending on all the requirements, these considerations may sometimes require developers to write functions specifically for the purpose. If this is necessary, the functions can often be simple "stub" functions that call existing functions in the LANSAs application.

Further considerations for functions calls executed through the LANSAs Composer Request Server

This Built-In Function will execute the function call through the LANSAs Composer request server.

When executed this way, the function call executes in another process or job (the request server). The Built-In Function and the request server process or job communicate cooperatively to execute the request and return the results.

This is generally transparent to your application. However, some special considerations apply to this mode of execution, including considerations related to:

- User profiles, authorities and execution environment
- IBM i work management (jobs and subsystems)
- The way in which the called function must be compiled

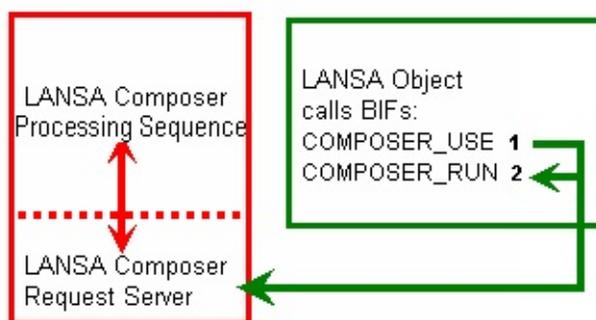
For information about requests executed through the LANSAs Composer request server, refer to *Appendix F (The LANSAs Composer Request Server)* in the *LANSAs Composer Guide*.

9.23 COMPOSER_RUN

⇒ **Note:** Built-In Function Rules.

COMPOSER_RUN runs a LANSА Composer Processing Sequence, in the LANSА Composer system identified by the server symbolic name argument, through the LANSА Composer Request Server. It can pass up to five named parameter values to the processing sequence.

COMPOSER_RUN assumes that the target system (the system identified by the LANSА Composer server symbolic name argument) is a LANSА Composer system greater than Version 3, and contains the LANSА Composer Request Server software. **This Built-In Function will fail if this is not the case.**



You must execute the [9.24 COMPOSER_USE](#) Built-In Function to define the server connection details and a symbolic name representing them before executing this Built-In Function.

Further important information about this Built-In Function is provided at the end of these notes.

For further information, also refer to the COMPOSER_RUN activity and *Appendix F - The LANSА Composer Request Server*, in the *LANSА Composer Guide*.

For use with

LANSА for i	YES
Visual LANSА for Windows	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	LANSAs Composer server symbolic name. The symbolic name is used to identify the connection details for the required LANSAs Composer server system. The name must have previously been specified in the COMPOSER_USE Built-In Function in the current session.	1	10		
2	A	Req	Processing Sequence identifier Identifies the LANSAs Composer processing sequence to run. Specify either the external identifier (name) or the internal identifier.	1	32		
3	A	Opt	Parameter name 1 (see Processing Sequence Parameters).	1	20		
4	A	Opt	Parameter value 1 (see Processing Sequence Parameters).	1	200		
5	A	Opt	Parameter name 2 (see Processing Sequence Parameters).	1	20		
6	A	Opt	Parameter value 2 (see Processing Sequence Parameters).	1	200		
7	A	Opt	Parameter name 3 (see Processing Sequence Parameters).	1	20		
8	A	Opt	Parameter value 3 (see Processing Sequence Parameters).	1	200		
9	A	Opt	Parameter name 4 (see Processing Sequence Parameters).	1	20		

10	A	Opt	Parameter value 4 (see Processing Sequence Parameters).	1	200		
11	A	Opt	Parameter name 5 (see Processing Sequence Parameters).	1	20		
12	A	Opt	Parameter value 5 (see Processing Sequence Parameters).	1	200		
13	A	Opt	Synchronous call? Specifies whether the Built-In Function waits for the processing sequence to complete. Defaults is 'Y', to wait. If any other value is specified, the Built-In Function posts the processing sequence run request and ends immediately.	1	1		
14	N	Opt	Synchronous time-out (seconds). The number of seconds the Built-In Function waits for a synchronous processing sequence run to complete. If the timeout is exceeded, the Built-In Function ends with a result code of 'TM'. Default is 30 seconds.	1	5	0	0
15	N	Opt	Request expires (seconds). IBM i servers only . Expiry does not apply to Winders servers. The number of seconds the request remains effective after it is posted to the request server. If more than the specified interval has elapsed before the request server begins to process the request, the request server will consider that the request has expired and will not process it. Default is zero (0). This means that no expiry applies to the request.	1	5	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Result code. OK, if the Built-In Function completes successfully. TM, if the synchronous request timed out. Any other result code signifies that an error occurred.	2	2		

Examples

This example uses a previously defined connection to a LANSa Composer system, with the symbolic name COMPOSER (see COMPOSER_USE), to call the processing sequence 'EXAMPLE_AATEST1'.

```
use builtin(COMPOSER_RUN) with_args('COMPOSER'
'EXAMPLE_AATEST1')
```

This example uses the same connection as above to call processing sequence 'EXAMPLE_AATEST2'. A single parameter: 'DIRECTORY' is passed with the value of '/'. The result code is received in variable #RESULT.

```
use builtin(COMPOSER_RUN) with_args('COMPOSER'
'EXAMPLE_AATEST2' 'DIRECTORY' '/') to_get(#RESULT)
```

This example uses the same connection as above to call processing sequence 'EXAMPLE_AATEST1', passing no parameters. The timeout value has been doubled from 30 seconds to 60 seconds.

```
Use Builtin(COMPOSER_RUN) With_Args('COMPOSER'
'EXAMPLE_AATEST1' *Default *Default *Default *Default *Default
*Default *Default *Default *Default *Default *Default 60) to_get(#RESULT)
```

Processing Sequence Parameters

The Built-In Function arguments can be used to pass up to five parameter values to the processing sequence run. A pair of Built-In Function arguments is used for each processing sequence parameter:

- The first argument in each pair (Parameter name n) should specify the parameter name as defined in the processing sequence to be run. If not specified, a default of '*NONE' is used which means that parameter pair is not used.
- The second argument in each pair (Parameter value n) should specify the value that is to be passed to the processing sequence for the corresponding parameter name. The maximum value length that can be passed is 200. If not specified, a default of '*NONE' is used. However, you should specify the parameter value for each parameter name that is specified.

Further considerations for processing sequences executed through the LANSAs Composer Request Server

This Built-In Function will run the processing sequence through the LANSAs Composer request server.

When executed this way, the processing sequence runs in another process or job (the request server). The Built-In Function and the request server process or job communicate cooperatively to execute the request and return the results.

This is generally transparent to your application. However, some special considerations apply to this mode of execution, including considerations related to:

- User profiles, authorities and execution environment
- IBM i work management (jobs and subsystems)

For information about requests executed through the LANSAs Composer request server, refer to *Appendix F (The LANSAs Composer Request Server)* in the *LANSAs Composer Guide*.

9.24 COMPOSER_USE

⇒ **Note:** Built-In Function Rules.

This Built-In Function associates a symbolic name with the details necessary for COMPOSER_CALLF or COMPOSER_RUN to connect to a nominated LANSa or LANSa Composer server system.

The association persists only for the duration of the current session or until COMPOSER_USE is used again to specify different LANSa Composer or LANSa Systems with the associated symbolic name.

The Result code of OK must be received before the LANSa Composer server symbolic name is used with the COMPOSER_RUN or COMPOSER_CALLF Built-In Functions.

Note that no connection is actually attempted until either the COMPOSER_RUN or COMPOSER_CALLF Built-In Functions is executed. Thus, connection errors arising from incorrect values used in this Built-In Function will not be evident until the connection is attempted by either the COMPOSER_RUN or COMPOSER_CALLF Built-In Functions.

For use with

LANSa for i	YES
Visual LANSa for Windows	YES
Visual LANSa for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	LANSa Composer server symbolic name. The symbolic name that will be used to identify the connection details specified. If successful, the symbolic name may be specified in subsequent invocations of the	1	10		

			COMPOSER_RUN and/or COMPOSER_CALLF Built-In Functions in the current session.				
2	A	Opt	Partition name. Specifies the name of the LANSAs partition in the specified LANSAs system in which LANSAs Composer is installed. 'LIC' is the default. This partition name is used in a standard LANSAs Composer installation.	3	3		

Arguments - IBM i server only

These arguments are ignored when running on a Windows server.

For a Windows Windows server, specify *DEFAULT for these arguments.

3	A	Opt	LANSAs program library name. (IBM i server only) Specifies the name of the LANSAs program library for the LANSAs system in which LANSAs Composer is installed. In a default LANSAs Composer installation, the library name would be LICPGMLIB. This is the default value used if this argument is not specified.	1	10		
4	A	Opt	Reserved for future use. Must be blank.	1	256		
5	A	Opt	Reserved for future use. Must be blank.	1	256		

Arguments - Windows server only

These arguments are ignored when running on an IBM i server system.

Note that although this Built-In Function allows for the long user names and passwords to be specified, the current version (3.0) of LANSAs Composer does not yet support the use long user names and passwords.

--	--	--	--	--	--	--	--

6	A	Opt	LANSA system path. (Windows server only) Specifies the path to the X_WIN95 folder in the LANSAsystem in which LANSAs Composer is installed: Default is C:\Program Files\LANSA Composer Server\X_WIN95 This would be the path in a default LANSAs Composer installation.	1256
7	A	Opt	User name. (Windows server only) Specifies the user name used to connect to the LANSAsystem in which LANSAs Composer is installed. (This value corresponds to the X_RUN parameter USER=.)	1256
8	A	Opt	Password. (Windows server only) Specifies the password used to connect to the LANSAsystem in which LANSAs Composer is installed. (This value corresponds to the X_RUN parameter PSPW=.)	1256
9	A	Opt	Data Source (Windows server only) Identifies the user database used with the LANSAsystem in which LANSAs Composer is installed. (This value corresponds to the X_RUN parameter DBID=.)	132
10	A	Opt	Database type. (Windows server only) Specifies the type of database specified in the previous argument. (This value corresponds to the X_RUN parameter DBUT=.)	120
11	A	Opt	Database user. (Windows server only) Specifies the user name for the database login, if required. (This value corresponds to the X_RUN parameter DBUS=.) It is 256 bytes long on Windows.	1256
12	A	Opt	Database password. (Windows server only) Specifies the password for the database login, if required. (This value corresponds to the X_RUN parameter DBPW=.) It is 256 byte long Windows.	1256

13	A	Opt	LANSA system overrides. (Windows server only) This argument may be used to specify a string of further X_RUN parameter names and values required to connect to the LANSAs system in which LANSAs Composer is installed. For more information, refer to The X_RUN Command .	1	128		
14	A	Opt	LANSA Composer Request Server logging enabled? (Windows server only) 'Y' if logging is enabled.	1	1		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Result code. If the Built-In Function completes successfully, the result code will contain 'OK'. Any other result code signifies that an error occurred.	2	2		

Examples

This example defines a connection with the symbolic name, COMPOSER, to the LANSAs or LANSAs Composer system in partition LIC (the default value) in the default LANSAs Composer installed location on either IBM i or Windows servers:

```
use builtin(COMPOSER_USE) with_args('COMPOSER')
```

This example defines a connection with the symbolic name ISERVER1 to partition 'PRD' in a LANSAs or LANSAs Composer system on the IBM i server that is executing the Built-In Function. The program library name for the target system is specified in variable #PGMLIB. The result code is received in variable #RESULT:

```
use builtin(COMPOSER_USE) with_args('ISERVER1' 'PRD' #PGMLIB)
to_get(#RESULT)
```

This example defines a connection with the symbolic name WINSERVER1 to partition 'PRD' in a LANSAs or LANSAs Composer system on the Windows server that is executing the Built-In Function. The path to the target system is specified in variable #XWIN95. Literal values have been used to specify the remaining connection details - for example, the user name and password is specified as 'PCXUSER', and the target system uses a Sybase SQL anywhere database with name LX_LANSAs. LANSAs Composer request server logging is enabled. The result code is received in variable #RESULT:

```
use builtin(COMPOSER_USE) with_args('WINSERVER1' 'PRD' *default
*default *default #XWIN95 'PCXUSER' 'PCXUSER' 'LX_LANSAs'
'SQLANYWHERE' 'DBA' 'SQL' *default 'Y') to_get(#RESULT)
```

9.25 CONCAT

⇒ **Note:** Built-In Function Rules.

Concatenates up to five alphanumeric strings to form one string as a return value. No truncation of trailing blanks is performed by this function.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	1st string to concatenate	1	Unlimited		
2	U	Req	2nd string to concatenate	1	Unlimited		
3	U	Opt	3rd string to concatenate	1	Unlimited		
4	U	Opt	4th string to concatenate	1	Unlimited		
5	U	Opt	5th string to concatenate	1	Unlimited		

Return Values

No	Type	Req/ Opt	Description	Min	Max Len	Min	Max
----	------	-------------	-------------	-----	---------	-----	-----

		Opt		Len		Dec	Dec
1	U	Req	Concatenated result string	1	Unlimited		
2	N	Opt	Length of returned string	1	15	0	0

Technical Notes

This Built-In function contains two consecutive operations:

1. Concatenate the input strings.

The length of the resulting string then is put into the second returning field (if it is provided). The concatenation goes through all the input strings unless the length of the resulting string is bigger than the maximum possible length of the first returning field type. In this case, the concatenation will stop and the second returning field will have the value of the maximum possible length of the first returning field type.

2. Assign the resulting string into the first returning field.

If the length of the returning field is smaller than the length of the resulting string, the last will be truncated.

Therefore, the value of the second returning field is not the length of the string in the first returning field.

For the maximum possible length of a field type please refer to [Field Type Considerations](#).

Examples

USE BUILTIN(CONCAT) WITH_ARGS(#VAL1 #VAL2 #VAL3 #VAL4 #VAL5) TO_GET(#VAL6 #SIGN150)

Example number	#VAL1 length	#VAL2 length	#VAL3 length	#VAL4 length	#VAL5 length	#VAL6 max length	#VAL6 type	#SIGN150 returned value
Example 1	2	2	2	2	2	7	Alpha	10

Example	256	2	2	2	2	7	Alpha	256
2								

In Example 1, all 5 input fields contain 2 characters strings. Even the #VAL6 can take up to only 7 bytes, the #SIGN150 gets back 10, which is 2 +2 +2 +2 +2.

In Example 2, the concatenation stops at #VAL2 because the resulting string length is bigger than 256, which is the maximum possible length of an Alphanumeric field (#VAL6 type). #The SIGN150 value is 256.

In both examples #VAL6 returns only 7 characters.

9.26 CONNECT_FILE

⇒ **Note:** Built-In Function Rules.

Prepares LANSAs so that all following I/O requests to the nominated physical file (and any views based on it) are rerouted to the server. Refer to [Database Connection](#) for more details.

The time taken to establish a file connection is very fast. It simply updates a routing table and does not communicate to the server at all.

The connection remains in effect until it is explicitly terminated by use of the Built-In Function DISCONNECT_FILE or by the ending of the LANSAs environment.

You should design your applications so that a minimal number of connection and disconnection points are used.

Different files may be connected to different server systems at the same time, but a single file cannot be connected to more than one server at a time.

The word "File" here refers to the base physical file (or table) and all logical files (or views) that are based on it.

A request to connect a file to a server that it is already connected to will be ignored, apart from resetting the selection block size and selection limit. No error will result.

A fatal error is caused by a request to connect a:

- file to a server that is different to the one that it is currently connected.
- LANSAs File (not an OTHER File or SQL View) that does not have AUTO_RRN set on to a database that requires an RRN path defined.

For use with

LANSAs for i	YES	Only available with RDMLX
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Physical File/Table Name. The name must be specified using uppercase characters. No check is made for the validity or existence of the name specified. The name may be specified as a generic name. The '*' symbol is used as the generic delimiter.	1	10		
2	A	Req	SSN of defined server.	1	10		
3	N	Opt	Selection Using the CONNECT_FILE Block Size . Default = 50. Note 1: For files used in SELECT loops with the RETURN_RRN parameter, you should use block size 1. Larger block sizes will result in unpredictable values being returned in the RETURN_RRN value. Note 2: For files used in SELECT loops and altered by DELETE or UPDATE commands that do not have WITH_KEY or WITH_RRN parameters (i.e. update or delete of the last record read) you should use block size 1. Note 3: If any of the fields to be selected is a BLOB or a CLOB and therefore might require a file to be send from the server to the client, the behavior will be as if you have used block size 1 for the SELECT process.	1	10	0	0
4	N	Opt	Selection limit. This option does not programmatically limit a selection loop to 'n' rows. It should not be used for this purpose. It is designed to stop a runaway (that is, out of control) select loop from attempting to transfer too much data.	1	10	0	0

		Exceeding the selection limit value will cause a fatal application error. The number of rows returned in this error situation is unspecified. Default = 10000				
--	--	--	--	--	--	--

Return Values

No return values.

Technical Notes

- Connecting a file while it is "in use" (e.g: in the middle of a SELECT loop when the file being selected is not connected to a server or connected to another server) will cause application failure and/or unpredictable results.
- You cannot, under any circumstances, connect the LANSA for i DC@Fnn internal database files to your application via this Built-In Function. This rule is not checked, but it should not be violated.
- The entire LANSA SuperServer facility does not support multi-membered files in any way, shape or form. You may be able to devise a strategy that will actually allow you to execute or call server functions that access multi-membered files, but you should remember that you are using an unsupported and totally IBM i dependent facility. When an IBM i based I/O module is invoked via this facility it opens the required file member(s) via the current library list (*LIBL) and as the IBM i logically first member (usually symbolically named *FIRST).

If you attempt to interleave a Client based function (using library *LIBL member *FIRST) and a Server based function (using a POINT command to a library and/or member), and both functions access the same file(s) you may cause the associated I/O module to fail with message IOM0033. This will happen regardless of any POINT commands present in the Client function. POINT commands are ignored in all Visual LANSA (i.e. Client) environments.

- It is very strongly recommended that all "connect" logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to

the server(s) that are being used.

- Attempting to connect a file that is already connected (to the same server) does not cause an error. When the file you are attempting to connect is already connected, the selection block and limit values are updated. This technique may be used to dynamically alter the selection block/limit values, but not while I/O operations are pending (e.g: within a select loop).
- Do not attempt to connect a blank file name.
- When using generic file names (e.g. LM*, GL*, *) be extremely careful not to overlap any generic names. Failure to observe this rule will cause unpredictable results. This rule means that name "*" (any name) can only be used by itself, as any other file name connected before or after the "*" will overlap with it.
- Message information routed from the server machine (in any form) arrives in a text format. It is displayed and accessible to RDML functions in the normal manner (e.g. GET_MESSAGE) as pure text. The message identifier and message file name details are not available for messages that have been routed from a server. You should not design client applications that rely on reading specific message identifiers from the applications message queue.

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application.

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
endif
```

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

Using the CONNECT_FILE Block Size

By default, the Block size parameter is the value 50. This means that 50 records

will be returned to the client end of the application for each trip to the server. The data will still be processed in the sequence in which it exists in the particular file, and debug will show the code looping through 50 records, but only one trip has been made to the server to retrieve the data.

Because of this, using either the relative record number or the last record read has inherent dangers.

Consider this code:

```
SELECT FIELDS(...) FROM_FILE(FILEA) RETURN_RRN(#RRN)
```

```
...
```

```
UPDATE FIELDS(...) IN_FILE(FILEA) WITH_RRN(#RRN)
```

```
ENDSELECT
```

This is acceptable on an IBM i server. However, in a client/server environment, the value of RRN after the select will be the relative record number of the 50th record. The OAM on the server has performed one read action, and therefore has returned the LAST RRN it is aware of.

The same applies to this code:

```
SELECT FIELDS(...) FROM_FILE(FILEA) RETURN_RRN(#RRN)
```

```
...
```

```
UPDATE FIELDS(...) IN_FILE(FILEA)
```

```
ENDSELECT
```

As far as the OAM is concerned, the last record read is the 50th record. Any attempted UPDATE or DELETE will be performed on the last record read.

Updating with a key to the file used in the SELECT loop is not allowed.

Therefore, the only possible solution is to set the block size to 1. This will ensure that the data is returned one record at a time.

The down side of this is that the performance of the application is significantly decreased. It should be noted at this point that the most efficient way to update multiple records on the server is to run the update code on the server using the CALL_SERVER_FUNCTION Built-In Function.

A similar issue occurs in maintenance applications that allow multiple detail records to be opened. For example, a Visual LANSA application displays a list of employees. The user can double click on an item in the list, and a maintenance form is opened. This receives the employee number from the list, and uses FETCH to read the data. FETCH causes the OAM to read only one record. Before saving the changes, the user opens another employee detail as well. The last record read value is stored in the OAM on the server, and as far as it is concerned, it was the second employee.

So, even though it was a FETCH that was performed, the employee maintenance form MUST update with a key. Any attempt to update the last record read will OVERWRITE the last record read.

9.27 CONNECT_SERVER

⇒ **Note:** Built-In Function Rules.

Connects an executing function to a server. The connection remains in effect until explicitly terminated by using the DISCONNECT_SERVER Built-In Function or by exiting from the Visual LANSA environment.

A server can be any SSN defined by one of the BIFs: DEFINE_OS_400_SERVER, DEFINE_ANY_SERVER, DEFINE_OTHER_SERVER and DEFINE_DB_SERVER. It can also be the special SSN *LOCALDB which refers to the local database server. Refer to [Database Connection](#) for more details.

The time taken to establish a connection is relatively long.

You should design your applications so that a single entry point connect is required rather than using many connects and disconnects.

A request to connect to a server that is already connected will be ignored. No error will result.

For use with

LANSA for i	YES	Only available with RDMLX
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	SSN of a defined server.	1	10		
2	A	Opt	Password to be used to connect to the server. This value is not stored and only exists for the duration of this function call. If this value is not specified, Application	1	256		

			<p>server connections use the value from the x_run parameter PSPW= for the default password.</p> <p>If this value is not specified, Database server connections use the password defined in the connection parameter overrides in DEFINE_DB_SERVER. You can let ODBC prompt for the password, as described in the 9.37 DEFINE_DB_SERVER example.</p>				
3	A	Opt	<p>If this value is 'Y', then the password value is ignored, and the authority under which the Windows application is running is used for authentication with the server via the Kerberos protocol.</p> <p>If this value is 'N', then the password argument is used for authentication.</p> <p>If this value is not specified, then the default value is the current setting of the PSTC parameter. Refer to The PSXX Parameters .</p>	1	1		
4	A	Opt	<p>User name to be used to connect to the server. This value is not stored and only exists for the duration of this function call.</p> <p>If this value is not specified, Application server connections use the value from the X_RUN parameter USER= for the default user name.</p> <p>This value is ignored in Database server connections.</p>	1	256		
5	A	Opt	<p>Handle Server Error</p> <p>If this value is set to 'Y', then the client X_Run session will not abort on errors returned from a Server. Instead the return code will be set to FE and server's error messages will be stored in the message queue.</p>	1	1		

		If set to 'N', the client X_Run session will abort on errors returned from a Server. Default Value is 'N'.				
--	--	---	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Connection established ER - Connection not established FE – Fatal error.	2	2		
2	A	Opt	Server Type. If a SuperServer, one of: AS400 (IBM i RPG) RDMLX400 (IBM i RDMLX) LINUX WINDOWS If a Database Server, one of: DB_ODBCORACLE DB_MSSQLS DB_MSACCESS DB_SQLANYWHERE DB_XXXXX where XXXXX is a user- defined database type Note: If the type cannot be determined, the server type will be returned as UNKNOWN.	3	15		
3	N	Opt	Connection Error Code. Zero if Return Code is OK or LANSAs is unable to determine an error code. If a SuperServer, this will be the Comms Error Code. The most common Error Codes	10	10	0	0

		are:				
		6				
		- Could not logon				
		17				
		- Unexpected error at client or server				
		20				
		- Could not locate server				
		If a Database Server, this may be the native database error, an ODBC/CLI API return code, or -9999 indicating an internal LANSAs error.				

Technical Notes

- It is very strongly recommended that all "connect" logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to the server(s) that are being used.
- By default this Built-In Function will connect to a partition on the server system that has the same identifier as the partition currently being used on the client system. For information on how to connect to a partition with a different identifier refer to the CONNECT_PARTITION option in the Built-In Function SET_SESSION_VALUE.
- When using Kerberos authentication to an IBM i server, the Kerberos Principal Name of the Windows domain user under whose authority the application is running, must be associated with a LANSAs User. This association is made using the IBM i Enterprise Identity Mapping (EIM) facility.

Technical Notes (IBM i Specific)

Kerberos works without further configuration directly to a server with no access outside that server, say to SQL Server or a file share.

If access outside of that one server is required – so called "multi-hop"

– then this is what is supported:

1. Trust whole computer to *any* services – We have tested and proved this is working
2. Trust a specific domain user to *any* services – We have tested and proved this is working (this requires setting up listener properly to run as a specific user, see the attached document. This should be verified first using lcoecho)

If your environment does not allow one of these configurations then multi-hop cannot be used.

Technical Notes (IBM i Specific)

- You may be concurrently connect to multiple different IBM i servers.
- You may be concurrently connected to the same IBM i server multiple times (with different SSN names).
- Database server connections are usually simpler to setup than communication servers are. The "first time" connection to communication servers is the most difficult. It is the time that causes frustration when a series of complex and often meaningless (without the Host Integration Server 2000 Error Major/Minor error codebook) error message numbers may appear
- The actual configuration and maintenance of communications between workstation clients and IBM i servers is actually beyond the scope of this reference guide, however the following tips and techniques may aid you in determining the cause of your problem.

Make sure that the user profile you are using is:

- Defined to IBM i.
- Authorized to use the LANSAPART partition required.
- 10 characters or less in length.
- Has a job description associated with it that has an initial library list that includes the LANSAPROG library (often DC@PGMLIB) and library QGPL.
- That the same job description has LOG(4 00 *SECLVL) logging if you are trying to solve a problem. This will ensure any IBM i job the profile starts will produce an IBM i job log. This job log will almost always yield useful error information.

- Enrolled in Office Services via the ADDDIRE and ADDOFCENR commands.
- Verify the IBM i user profile being used by signing on at a dumb terminal and then immediately typing a LANSA PARTITION(XXX) command. Does anything surprising happen?

Remove the cause of any errors or authority problems that are apparent before trying the profile through a communications link.

- Use the IBM i command WRKSBSJOB QCMN to display the active jobs in subsystem QCMN. Repeatedly use F5=Refresh, to refresh the list as the communication job starts.
Is subsystem QCMN active?

NO: Start it and try the operation again

YES: Does an IBM i "server" job appear in the QCMN subsystem?

If Immediately use "5=Work with" to trace this Job. Wait until it completes
YES and then check its resulting spooled job log file for details. If it does not produce a log, alter the job description associated with the user profile you are using to LOG(4 00 *SECLVL) until you trace the cause of the problem.

Do you have an "LXX" (i.e. LANSA SuperServer) license installed in the IBM i system? Is it current? Use the command LANSA REQUEST(LICENSE) to check.

If Have you started communications manager, PC Support, or other
NO communications router on the PC?

Does the communications manager "Display Messages" option show any error information?

Does the IBM i DSPLOG (display log) command show any communications error information?

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application.

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
endif
```

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

9.28 CONVERTDATE

⇒ **Note:** Built-In Function Rules.

Converts format of alphanumeric date.

Using the CONVERTDATE Built-In Function twice and producing 2 different result fields allows you to produce dates such as: THURSDAY 5TH OCTOBER 1987.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Date that is to be converted	1	20		
2	A	Req	Format of date to be converted	1	1		
3	A	Req	Format required of date in return value	1	1		

Valid Date Formats

- Date to be converted: A, B, C, D, E, F, G, H, I, J, K, L, M, V, W, X, Y, Z and 1.
- Date to be returned: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z and 1.
- Refer to [Date Formats](#)

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Returned converted date	1	20		
2	A	Opt	Return code (Y or N) for successful conversion	1	1		

Technical Notes

- All dates must have a four character year so that accurate comparisons and calculations can be performed. Where a two character year (e.g. DDMMYY, YYMMDD, MMY) is supplied the century value is retrieved from the system definition data area. The year supplied is compared to a year in the data area, if the supplied year is less than or equal to the comparison year then the less than century is used. If the supplied year, is greater than the comparison year, then the greater than century is used.
- When using date formats P, Q, R, S, T, U, the date is returned in the format specified in messages BIF0101 and BIF0102 in DC@M01. To have the date returned in a language other than English you should ensure these messages are translated into the appropriate language.

If LANG is something other than ENG or NAT, you will need to ensure the messages exist in the message file for the language you are executing in.

Example

Convert a date field #YMD in date format YYMMDD (D) to date format DDMMYY (B) in field #DMY:

```
USE      BUILTIN(CONVERTDATE) WITH_ARGS(#YMD D B) TO_GET(
```

9.29 CONVERTDATE_NUMERIC

⇒ **Note:** Built-In Function Rules.

Converts format of numeric date.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Date that is to be converted	4	8	0	0
2	A	Req	Format of date to be converted	1	1		
3	A	Req	Format required of date in return value	1	1		

Valid Date Formats

- Valid formats of the date to be converted: A, B, D, F, H, J, L, V, W, X, Y, Z and 1.
- Valid formats of the date to be returned: A, B, D, F, H, J, L, V, W, X, Y, Z and 1.
- Refer to [Date Formats](#)

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
----	------	-------------	-------------	------------	------------	------------	------------

1	N	Req	Returned converted date	4	8	0	0
2	A	Opt	Return code (Y or N) for successful conversion	1	1		

Technical Notes

All dates must have a four digit year

So that accurate comparisons and calculations can be performed, all dates must have a four digit year. Where a two digit year (e.g. DDMMYY, YYMMDD, MMYYY) is supplied the century value is retrieved from the system definition data area. The year supplied is compared to a year in the data area, if the supplied year is less than or equal to the comparison year then the less than century is used. If the supplied year is greater than the comparison year then the greater than century is used.

First argument and return value specifics

For the first argument and the first return Value, INTEGER and FLOAT fields cannot be used.

The DATE field cannot be used directly. However, the DATE field can be used indirectly when being converted into Number using the .AsNumber Intrinsic Function.

The value of the second argument should be in sync with the format that is put into the Intrinsic Function.

For example if you have a DATE field called DATEFL1, and you want to use it in the Built-In Function as the first argument, you must do a conversion like this:

```
DATEFL1.AsNumber(DDMMCCYY)
```

Please refer to the Intrinsic Function for other formats

The value of the second argument should be H, which indicates the DDMMYYYYY date format.

To hold the first return value, any NUMERIC field can be used including INTEGER and FLOAT.

Translations

When using date formats P, Q, R, S, T, U, the date is returned in the format specified in messages BIF0101 and BIF0102 in DC@M01. To have the date returned in a language other than English you should ensure these messages are translated into the appropriate language.

If LANG is something other than ENG or NAT, you will need to ensure the messages exist in the message file for the language you are executing in.

Example RDMLX only:

Use CONVERTDATE_NUMERIC with DATE field type only.

```
USE BUILTIN(CONVERTDATE_NUMERIC)
WITH_ARGS(#DATEFL1.AsNumber(DDMMCCYY) H
J)TO_GET(#NUM80 )
```

The following code must be used for a better programming style:

```
#NUM80 := #DATEFL1. AsNumber (CCYYMMDD)
#NUM80 is a numeric signed 8 bytes long ,0 decimal .
```

Example RDML

Convert a date field #YMD in date format YYMMDD (D) to date format DDMMYY (B) in field #DMY:

```
USE BUILTIN(CONVERTDATE_NUMERIC) WITH_ARGS(#YMD D I
```

9.30 CONVERT_STRING

⇒ **Note:** Built-In Function Rules.

This Built-In Function converts a text string from one encoding to another.

For use with

LANSA for i	YES
Visual LANSA for Windows	Not applicable
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be converted.	1	Unlimited	0	0
2	N	Req	To code page.	1	5	0	0
3	N	Opt	From code page. Default: code page of the current job.	1	5	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Converted string returned.	1	Unlimited	0	0
2	A	Opt	Return code. OK = conversion completed. ER = An error occurred.	2	2	0	

Technical Notes

This BIF is intended for IBM i use. The IBM i translation table for the 'from code' to the 'to code' page must exist. If the table does not exist, the returned string will be the same as for the argument string.

If this Built-In Function is executed in Visual LANSA for Windows, no conversion will be done. The returned string will be the same as the argument string.

Example

```
FUNCTION  OPTIONS(*DIRECT)
DEFINE   FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
DEFINE   FIELD(#tocode) TYPE(*DEC) LENGTH(5) DECIMALS(0)
DEFINE   FIELD(#fromcode) TYPE(*DEC) LENGTH(5) DECIMALS(0)
DEFINE   FIELD(#String1) TYPE(*CHAR) LENGTH(256) INPUT_ATR(LC)
DEFINE   FIELD(#string2) TYPE(*CHAR) LENGTH(256) INPUT_ATR(LC)
REQUEST  FIELDS(#string1 #tocode #fromcode)
USE      BUILTIN(convert_string) WITH_ARGS(#string1 #tocode #fromcode)
DISPLAY  FIELDS((#RETCOD) (#string2))
```

9.31 CREATE_SPACE

⇒ **Note:** Built-In Function Rules.

Creates a space object with the specified name.

For use with

LANSA for i	YES	Only available with RDMLX
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name A name must be specified. Names are NOT case sensitive. Names should not be started with an asterisk (*) or blank character.	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "OK" = Space created or already exists. "ER" = Creation attempt failed. Messages issued will indicate more about the cause of the failure.	2	2		

Technical Notes

Space objects are unique within an operating system process (or job) by their name (case insensitive).

Space objects cannot be shared between operating system processes (or jobs). If operating system process A and B are both using a space object named X then they each have their own unique and independent instance of the space object named X.

Space objects are not persistent. A space object and its data content ceases to exist when the operating system process (or job) that owns them ends.

Space Naming Recommendations and Techniques

In most simple RDML function contexts a naming convention based on the name of the current function should be used.

This is demonstrated in this working function:

```
FUNCTION OPTIONS(*DIRECT)
***** COMMENT(Prototypical cells must be referenced)
GROUP_BY NAME(#XG_SPACE) FIELDS(#EMPNO #SURNAME #GIVEI
***** COMMENT(Space name is based on function name and default)
DEFINE FIELD(#SPACENAME) REFFLD(#FUNCTION)
***** COMMENT(Create space keyed cell EMPNO and with 2 data cel
USE BUILTIN(CREATE_SPACE) WITH_ARGS(#SPACENAME)
USE BUILTIN(DEFINE_SPACE_CELL) WITH_ARGS(#SPACENAME EMI
USE BUILTIN(DEFINE_SPACE_CELL) WITH_ARGS(#SPACENAME GIV
USE BUILTIN(DEFINE_SPACE_CELL) WITH_ARGS(#SPACENAME SUF
***** COMMENT(Don't forget to destroy the space when finished)
USE BUILTIN(DESTROY_SPACE) WITH_ARGS(#SPACENAME)
```

Where a form or function creates multiple spaces an approach that suffixes the space names with a unique identifier is recommended.

This code fragment demonstrates a technique for doing this:

```
***** COMMENT(Name is longer and contains the function name)
DEFINE FIELD(#EMPSPACE) REFFLD(#SYSVAR$AV) DEFAULT(*FUNC
DEFINE FIELD(#DEPSPACE) REFFLD(#SYSVAR$AV) DEFAULT(*FUNC
DEFINE FIELD(#SECSPACE) REFFLD(#SYSVAR$AV) DEFAULT(*FUNC
***** COMMENT(During initialization/startup)
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#EMPSPACE '.Emp') TO_GET(#F
USE BUILTIN(TCONCAT) WITH_ARGS(#DEPSPACE '.Dep') TO_GET(#D
USE BUILTIN(TCONCAT) WITH_ARGS(#SECSPACE '.Sec') TO_GET(#SE
```

(In a component you would of course use a default value of *COMPONENT rather than *FUNCTION for each of the space names).

In more complex environments there is a possibility that a space has already been created and defined by another instance or previous execution of the same piece of logic.

In such situations you can check whether a space exists before attempting to create it by using a technique like the one demonstrated in these RDMLX code fragments:

```
Define #SpaceName RefFld(#SysVar$Av) Default(*Component)
Define #SpaceRC *Char 2
Def_Cond *NoSpace '#SpaceRC *ne OK'

EvtRoutine Handling(#Com_Owner.CreateInstance)
Use Space_Operation (#SpaceName CheckExistence) (#SpaceRC)
If *NoSpace
Use Create_Space (#SpaceName)
Use Define_Space_Cell (#SpaceName EmpNo Key)
Use Define_Space_Cell (#SpaceName SurName)
Use Define_Space_Cell (#SpaceName GiveName)
Endif
EndRoutine
```

In more complex multi-instance RDMLX components you may sometimes require a unique space (remembering that spaces are unique by name) for each separate instance of the component.

A technique like this may be used:

```
Define #SpaceName RefFld(#SysVar$Av) Default(*Component)
Define #SpaceRC *Char 2
Def_Cond *NoSpace '#SpaceRC *ne OK'
EvtRoutine Handling(#Com_Owner.CreateInstance)
Invoke #Com_Owner.CreateUniqueSpace ReturnName(#SpaceName)
Use Define_Space_Cell (#SpaceName EmpNo Key)
Use Define_Space_Cell (#SpaceName SurName)
```

```

Use Define_Space_Cell (#SpaceName GiveName)
EndRoutine
MthRoutine CreateUniqueSpace
Define_Map *Output #SysVar$av #ReturnName
Define #TempName RefFld(#SpaceName)
Define #TempChar *Char 10
Define #TempNum RefFld(#Date) Length(10) Decimals(0) edit_code(4) defau
Begin_Loop Using(#TempNum)
Use TConcat (*Component '.' #TempChar) (#TempName)
Use Space_Operation (#TempName CheckExistence) (#SpaceRC)
Leave *NoSpace
End_Loop
Use Create_Space (#TempName)
Set #ReturnName Value(#TempName)
EndRoutine

```

Other Tips, Techniques and Recommendations

In high volume repeated commands avoid using visually defined fields as mapping values unless absolutely necessary. When a field has been visually defined mapping into or out of its value is significantly slower because of the underlying visual context.

For example, imagine that you want to count the total number of rows in an employee space (which has about 125,000 rows).

You might use code like this:

```

Def_Cond *Okay '#SpaceRC = OK'
Change #EmpTotal 0
Use Select_in_Space #Space (#SpaceRc #EmpNo #GiveName #SurName)
DoWhile *okay
Change #EmpTotal '#EmpTotal + 1'
Use SelectNext_in_Space #Space (#SpaceRc #EmpNo #GiveName #SurName)
EndWhile

```

If any one of the fields #Space, #SpaceRc, #EmpNo, #GiveName or #SurName is defined in a visual context (ie: as part of a DEFINE_COM) then the performance of this loop will be impacted by mapping values into them 125,000 times.

Assuming that #EmpNo, #GiveName or #SurName are in fact defined in a

visual context then to improve the performance of this logic you could do this:

```
Def_Cond *Okay '#SpaceRc = OK'  
Change (EmpTotal 0  
Use Select_in_Space #Space (#SpaceRc)  
DoWhile *okay  
Change #EmpTotal '#EmpTotal + 1'  
Use SelectNext_in_Space #Space (#SpaceRc)  
EndWhile
```

Or do this:

```
Def_Cond *Okay '#SpaceRc = OK'  
Define #XEmpNo RefFld(#EmpNo)  
Define #XGiveName RefFld(#GiveName)  
Define #XSurName RefFld(#SurName);  
Change (EmpTotal 0  
Use Select_in_Space #Space #SpaceRc #XEmpNo #XGiveName #XSurName  
DoWhile *okay  
Change #EmpTotal '#EmpTotal + 1'  
Use SelectNext_in_Space #Space (#SpaceRc #XEmpNo #XGiveName #XSurName  
EndWhile
```

Example

This example defines a space whose name is the current components name suffixed by ".emp" and then defines 3 cells within it whose type and length are based on the definitions of fields EMPNO, GIVENAME and SURNAME respectively. The first cell the key to the space:

```
Define #SpaceName *char 20  
Use TConcat (*component '.EMP') (#SpaceName)  
Use Create_Space (#SpaceName)  
Use Define_Space_Cell (#SpaceName EmpNo Key)  
Use Define_Space_Cell (#SpaceName GiveName)  
Use Define_Space_Cell (#SpaceName SurName)
```

9.32 CREATE_PROMPT_FILE

⇒ **Note:** Built-In Function Rules.

This Built-In Function is used to create a physical file containing all Promptable File Library Variables extracted from the importing package in a runtime environment. The generated file can be used to update the variables' values before calling the IMPORT_OBJECTS Built-In Function.

The generated file is only valid to the IMPORT_OBJECTS Built-In Function when the CREATE_PROMPT_FILE and IMPORT_OBJECTS are in the same session.

For use with

LANSA for i	No
Visual LANSA for Windows	Yes
Visual LANSA for Linux	No

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Importing Package Folder	1	256		
2	A	Req	Requested File Type. The supported file type values are: *ImportLibraryVar: Import substitution variables.	1	20		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
----	------	-------------	-------------	------------	------------	------------	------------

1	A	Req	Return values are: OK : A file is generated. NR : No promptable variable found. No file is created. ER : Invalid argument or invalid environment. For example, ER is returned for an IBM i.	2	2		
2	A	O	The newly created file.	1	256		

9.33 DATEDIFFERENCE

⇒ **Note:** Built-In Function Rules.

Calculates the difference between two given dates in number of days. The return code indicates if the format of the dates or the dates themselves are valid (Y or N). The sign of the calculated value may also be returned.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Start/First date. See Note.	6	8	0	0
2	A	Req	Format of Start/First Date	1	1		
3	N	Req	End/Second date. See Note.	6	8	0	0
4	A	Req	Format of End/Second Date	1	1		

Valid Date Formats

- Valid formats for first and second dates: A, B, D, F, H, J, L, V and 1.
- Refer to [Date Formats](#)

Note:

For the first and the third argument, the INTEGER and FLOAT field cannot be used.

The DATE field can be used, but it should first be converted into string using the .AsNumber Intrinsic Function. For further information, please refer to the *What's New in V11.0* list for information about Intrinsic.

The value of the second and the fourth argument must be in sync with the format that is put into the intrinsic function.

For example, if you have a DATE field called DATEFL1, and want to use it in the BIF as the first argument, you must do a conversion like this:

DATEFL1.AsNumber(DDMMCCYY) (Please refer to Intrinsic Function for other formats.)

In this example, the value of the second argument should be H, which indicates the DDMMYYYY date format.

To hold the first return value, any NUMERIC field can be used including INTEGER and FLOAT.

Example RDMLX only:

```
USE BUILTIN(DATEDIFFERENCE_ALPHA)
WITH_ARGS(#DATEFL1. AsNumber(DDMMCCYY) H #DATEFL2.
AsNumber (CCYYMMDD) J)TO_GET(#DEC80 )
```

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Difference (beg to end) in days	3	8	0	0
2	A	Opt	Sign of difference (+,-)	1	1		
3	A	Opt	Return code (Y, N) for complete.	1	1		

Note: All dates must have a four character year so that accurate comparisons and calculations can be performed. Where a two character year (e.g. DDMMYY, YYMMDD, MMY) is supplied the century value is retrieved

from the system definition data area. The year supplied is compared to a year in the data area, if the supplied year is less than or equal to the comparison year then the less than century is used. If the supplied year is greater than the comparison year then the greater than century is used.

Example

Calculate the difference #DIFF in days between date field #YMD in date format YYMMDD (D) and date field #DMY in date format DDMMYY (B):

```
USE      BUILTIN(DATEDIFFERENCE) WITH_ARGS(#YMD D #DMY B)
        TO_GET(#DIFF #SIGN)
```

9.34 DATEDIFFERENCE_ALPHA

⇒ **Note:** Built-In Function Rules.

Calculates the difference between two given dates in number of days. The return code indicates if the format of the dates or the dates themselves are valid (Y or N). The sign of the calculated value may also be returned.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Start/First date. See Note.	1	10		
2	A	Req	Format of Start/First Date	1	1		
3	A	Req	End/Second date See Note.	1	10		
4	A	Req	Format of End/Second Date	1	1		

Valid Date Formats

- Valid formats for first and second dates: A, B, C, D, E, F, G, H, I, J, K, L, M, V and 1.
- Refer to [Date Formats](#)

Note:

For the first and the third argument, the INTEGER and FLOAT field cannot be used.

The DATE field can be used, but it should first be converted into string using the .AsNumber Intrinsic Function. For further information, please refer to the *What's New in V11.0* list for information about Ininsics.

The value of the second and the fourth argument must be in sync with the format that is put into the intrinsic function.

For example, if you have a DATE field called DATEFL1, and want to use it in the BIF as the first argument, you must do a conversion like this:

DATEFL1.AsNumber(DDMMCCYY) (Please refer to Intrinsic Function for other formats.)

In this example, the value of the second argument should be H, which indicates the DDMMYYYY date format.

To hold the first return value, any NUMERIC field can be used including INTEGER and FLOAT.

Example RDMLX only:

```
USE BUILTIN(DATEDIFFERENCE_ALPHA)
WITH_ARGS(#DATEFL1.AsNumber(DDMMCCYY) H #DATEFL2.
AsNumber (CCYYMMDD) J)TO_GET(#DEC80 )
```

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Difference (beg to end) in days	3	8	0	0
2	A	Opt	Sign of difference (+,-)	1	1		
3	A	Opt	Return code (Y, N) for complete.	1	1		

Note: All dates must have a four character year so that accurate comparisons and calculations can be performed. Where a two character year (e.g. DDMMYY, YYMMDD, MMY) is supplied the century value is retrieved

from the system definition data area. The year supplied is compared to a year in the data area, if the supplied year is less than or equal to the comparison year then the less than century is used. If the supplied year is greater than the comparison year then the greater than century is used.

Example

Calculate the difference #DIFF in days between date field #YMD in date format YYMMDD (D) and date field #DMY in date format DDMMYY (B):

```
USE      BUILTIN(DATEDIFFERENCE_ALPHA) WITH_ARGS(#YMD D #  
          TO_GET(#DIFF #SIGN)
```

9.35 DECRYPT

⇒ **Note:** Built-In Function Rules.

Decrypt a text string.

This Built-in Function is used in conjunction with [9.79 ENCRYPT](#).

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Text to be decrypted	8	Unlimited		
2	N	Req	Length of expected output text after decryption. This is the same value as that used for the second argument, <i>length of text to be encrypted</i> , during encryption. When encrypted text is stored in HEX (the recommended way to store any encrypted text) this value is half the length of the text to be decrypted. The value for this length argument must be a multiple of 8. The value provided for this argument must not be greater than the length of Argument 1 (text to be decrypted)	1	11	0	0
3	u	Req	Key to be used for decryption This key must be the same key as was	16	32		

			used with the ENCRYPT function.				
4	A	Opt	Encrypted text stored in HEX. YES=Text to be decrypted is in HEX format. NO= Text to be decrypted is in binary format. Default is NO	2	3		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	u	Req	Returned decrypted text	8	Unlimited		
2	A	Opt	Return code OK = action completed ER = error occurred	2	2		

Technical Notes

Refer to [9.79 ENCRYPT](#).

Example

Refer to [9.79 ENCRYPT](#).

9.36 DEFINE_ANY_SERVER

⇒ **Note:** Built-In Function Rules.

Defines the details of a LANSAsystem that is to be used as a Server.

The definition details are not persistent and only exist while the LANSAsystem environment remains active.

The time taken to define a server is minimal.

Platform Considerations

IBM i Server This Built-In Function must be used for I/O commands to RDMLX files or to call an RDMLX function on the server. It may also be used instead of DEFINE_OS_400_SERVER to access non-RDMLX objects, if they have been recompiled since enabling the partition for RDMLX.

If the partition has not been enabled for RDMLX or the non-RDMLX objects have not been recompiled, DEFINE_OS_400_SERVER must be used to access non-RDMLX objects.

Other Servers Either this Built-In Function or DEFINE_OTHER_SERVER may be used to access any object on the server.

For use with

LANSAs for i	YES	Only available with RDMLX
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	SSN (Server Symbolic Name) This is the name that will be used in all future references to this server by this and other	1	10		0

			RDML functions.				
2	A	Req	LU Partner Name	1	20		0
3	A	Req	Ignored on non-IBM i Servers (Always Y) Y or 1 - Start commitment control on the server. Once this server is connected it will receive a "commit" or "rollback" request whenever the client issues an RDML level COMMIT or ROLLBACK command. other - do not use commitment control	1	1		
4	A	Opt	X_RUN exceptional arguments.	1	256		0
5	A	Opt	Divert LOCK_OBJECT requests to this server. If this option is used all subsequent LOCK_OBJECT requests will be diverted to this server. Multiple servers will receive the same LOCK_OBJECT request if multiple servers have this option enabled concurrently. In such cases, a lock must be granted on all participating servers for the LOCK_OBJECT to complete normally. Where one server fails to grant the lock an UNLOCK_OBJECT request is made to all servers that have already granted the object lock. Note that *AUTONUM, *AUTOALP, and *DTAssslllxxxxxxxxxxx system variables are also retrieved from the server if locks are diverted to the server. Refer to *AUTONUM and *AUTOALP System Variables (Data Areas) and *DTASSLLLXXXXXXXXXX System Variables (Data Areas) in the <i>LANSA Application Design Guide</i> .	1	1		0

			<p>Y or 1 - Divert LOCK_OBJECT requests. Z - Divert LOCK_OBJECT requests and also divert authority checking requests to this server (only one server should be nominated as the diversion target for authority checking requests). R - Route lock requests AND authority requests AND repository data requests (if not found locally). Refer also to the Notes of the X_RUN parameter PSRR. Other- do not divert LOCK_OBJECT requests. The default value is Z.</p>			
6	A	Opt	<p>Show "Please Wait" message while connecting. Y or 1 - Show wait message. Other - do not show message. Default value is Y.</p>	1	1	0
7	A	Opt	<p>Ignored on non-IBM i Servers Execution priority. Default value is '20'. Specify other values as per the IBM i command CHGJOB parameter RUNPTY. User should be authorized to change to the nominated value.</p>	1	2	
8	A	Opt	<p>Ignored on non-IBM i Servers Client-to-Server conversion table name to be used. No library name can be specified Defaults to *JOB, meaning the translation table will be generated based on the client code page and the IBM i server job's CCSID. If this argument is *JOB then the Server-to-Client table must also be *JOB.</p>	1	10	0
9	A	Opt	<p>Ignored on non-IBM i Servers Server-to-Client conversion table name to</p>	1	10	0

		<p>be used. No library name can be specified. Defaults to *JOB, meaning the translation table will be generated based on the client code page and the IBM i server job's CCSID.</p> <p>If this argument is *JOB then the Client-to-Server table must also be *JOB.</p>				
--	--	--	--	--	--	--

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code: OK - Server Defined	2	2		0

Technical Notes

- To use this BIF you must set x_run parameter CDLL to LCOMGR32.DLL and x_run parameter CMTH must be C or T.
- The Server Network Name you specify when invoking this Built-In Function should be identical to the Partner LU Name under which the server was (or will be) enrolled within the LANSAs Communications Administrator.
- It is very strongly recommended that your server definition and connection logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to the server(s) that are being used.
- It is recommended that you use SSN values that are meaningful to end users (e.g: CHICAGO, BOSTON, CHARLIE1, etc.) as they may appear in messages from time to time.
- SSN names must be unique and start with a character in the English language alphabet (uppercase A through Z).
- A server may be repeatedly defined and (re)defined when it is not connected. If you attempt to (re)define a server that is currently connected a fatal error will result.

- The X_RUN exceptional argument may be used to override the parameters used on the X_RUN command started on the server system.

By default, the following client X_RUN parameter values are passed to (and inherited by) the X_RUN command started on the server system: CMTH=, CDLL=, DATF=, DATS=, DBCF=, DBCL=, DBLK=, DBTC=, DBUS=, DEVE=, FXQF=, FXQM=, HSKC=, INIT=, ITHP=, ITRC=, ITRL=, ITRM=, ITRO=, LANG=, PART=, PRTR=, PSPW=, PSTC=, PSWD=, TASK=, TERM=, USER=, XAFP= and XCMD=. The following client X_RUN parameter values are only inherited by the server if the client and the server are using the same operating system: DBID=, DBII=, DBIT=, DBUT=, ODBI= and WPEN (and related Windows Printing Extension [parameters](#)).

All other X_RUN parameter values on the server system are defaulted (on the server system) in the usual manner (that is, from a profile file, from system environment settings, and so on). Refer to the definition of the X_RUN command for details of all parameter values and the methods by which they can be specified and defaulted.

- You may override any server X_RUN parameter (via the X_RUN exceptional argument value) except for CDLL=, CMTH=, DATF=, DATS=, DEBUG=, DEVE=, LANG=, MODE=, PART=, PROC=, PSPW=, USER= and XAFP=. These X_RUN arguments are unconditionally inherited from the client system. However, some of these parameters may be altered by calling [SET_SESSION_VALUE](#) before invoking CONNECT_SERVER.

Override parameters may be given a specific value, or the special value *SERVER, which indicates that the server default should be used. As an example, a Windows client using DBII=*NONE might connect to a Windows Server running Oracle. By default, Windows uses the database type MSSQLS (SQL Server), so DBUT needs to be overridden. The X_RUN exceptional argument value could be set to either DBUT=ODBCORACLE or DBUT=*SERVER.

- The details defined via this Built-In Function are not persistent. They are lost when the X_RUN command completes. You may choose to define your own set of SQL based tables to hold server details and actually read the table(s) to get values to be passed on to this Built-In Function.
- Please experiment with these facilities first and then design some sort of server architecture for your organization that has these characteristics:

- It matches your organization's requirements.
- It is quick and easy to change.
- It is extensible.

Do this before launching into any large-scale design or development project.

- The client's date format will be automatically passed to the server. If the date formats are different (e.g. MDY vs DMY), the server will automatically return data in the client's format.

The client's date format can be changed from the default by specifying the `x_run` parameter `DATF=`. Please refer to [Standard X_RUN Parameters](#) for more information about this parameter.

Note that if the client and server date formats are different, Date format validation rules specifying exact formats (e.g. DDMMYY) will fail (as the data may be returned as MMDDYY). Date format SYSFMT is recommended where clients need to use different date formats (e.g. USA and UK clients).

Notes on Commitment Control

- If Start Commitment Control is Y, LANSAs will automatically start and end commitment control. See [User Exit F@BGNCMT - Start Commitment Control](#) and [User Exit F@ENDCMT - End Commitment Control](#) in the *LANSAs for i User Guide* for details.
- When the server has been indicated as having commitment control started, it will effect all subsequent COMMIT and ROLLBACK commands issued. When a COMMIT or ROLLBACK command is issued the routine involved loops through all currently connected servers. To each one that has commitment control active, it issues a "commit" or "rollback" request and then waits for the server to respond before proceeding. This is done after a commit/rollback has been issued correctly to the local/client database management system.

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application:

```
if (#retcode *ne OK)
```

```
abort msgtxt('Failed to .....')
```

```
endif
```

Let the standard error handling built into every generated application, take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

DBCS Considerations

When the server is indicated as being DBCS capable an additional translation table must be locatable on the client PC.

This table must be named X_CT<language code>.DAT and must be located in the X_LANSA\EXECUTE directory.

A version of this table named X_CTJPN.DAT for <language code> "JPN" (Japanese) is available to all Japanese customers.

This translation file contains the following notes:

- This table is shipped in an "as is" condition to support customer tailoring. No warranty is expressed or implied. It is the customer's responsibility to maintain and verify this table.
- This table is loaded by the Visual LANSA DEFINE_ANY_SERVER function when the current language has DBCS capability.
- The name of the table loaded is derived from "x_ct" combined with the current language code (e.g. jpn) and the suffix ".dat". Thus if language code jpn was being used, the table name would have to be "x_ctjpn.dat". For language code tchi the table name would be "x_cttchi.dat".
- The table must reside in the <drive>:\x_lansa\execute directory where <drive> is whatever local or server disk drive onto which Visual LANSA has been installed.
- This conversion table is only used for the double byte parts of any string. Single byte parts of any string at all (DBCS allowed or not) are always converted by using the IBM i single byte tables specified in the DEFINE_ANY_SERVER Built-In Function call.
- This means that a field containing mixed double and single byte characters is partially converted by this table and partially by the single byte conversion

table that is used by both DBCS and non-DBCS conversions.

- DBCS conversion of data within an individual field only occurs when the field is indicated as being DBCS capable (e.g. dictionary keyboard attributes j, e, o, etc.) and when the current language is DBCS capable. If both these conditions are not met the entire field is converted as a single byte string by the single byte conversion tables previously mentioned.
- An * in column 1 indicates a comment line.
- All values are specified in hexadecimal format.

Separate values with a single comma (,) only.

9.37 DEFINE_DB_SERVER

⇒ **Note:** Built-In Function Rules.

Defines details of a database that is to be used for files that are specifically redirected to use this database.

Primarily, this is for the purpose of supporting Other Files and SQL Views in Other databases. Other Files and SQL Views are loaded into LANSAs using Load Other Files on the File Control menu.

It can also be used to access LANSAs User Files in a different partition or in an external database. These file definitions must be exported from the repository for the external database into the current repository. Whenever the file definition is changed in the external repository it will need to be re-imported to the current repository and the OAM re-built.

The definition details are not persistent and only exist while the LANSAs environment remains active.

The time taken to define a database is minimal.

Note 1: This BIF has a similar purpose to DEFINE_OTHER_SERVER except that the OAMs still reside on the local PC and database I/O is used to access the server. See [Database Connection](#).

Note 2: This BIF requires a local database connection established, otherwise it will return an error.

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	SSN (Server Symbolic Name) This is the	1	10		

			name that will be used in all future references to this server by this and other RDML functions.				
2	A	Req	Database Identifier.	1	32		
3	A	Opt	Connection Parameter Overrides (may be blank).	1	256		
4	A	Opt	RRN Path Override (may be blank). If it is specified, it must end with a path separator - on MS Windows, this is a back slash.	1	256		
5	A	Opt	Database Type Override (may be blank). Specify a valid database type from The X_DBMENV.DAT File .	1	32		
6	A	Opt	ODBI= Override to specifically set the transaction isolation level for this database. May be 0-4 or blank. 0 means the default setting for the database. Blank means the same as the ODBI= parameter. For more information, refer to The ODBI=Parameter .	1	1		
7	A	Opt	ODBA= This parameter has been deprecated. For more information, refer to The ODBA=Parameter .	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Server Defined ER - Server not defined and error message(s) issued.	2	2		

Technical Notes

- The Database Identifier must be a valid ODBC database name, defined in the 32-bit ODBC Administrator. It would normally be the same as the ODBC database name that was used to load the OTHER File into LANSAs.
- It is very strongly recommended that your server definition and connection logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to the server(s) that are being used.
- It is recommended that you use SSN values that are meaningful to end users (e.g. CHICAGO, BOSTON, CHARLIE1, etc.) as they may appear in messages from time to time.
- SSN names must be unique and start with a character in the English language alphabet (uppercase 'A' through 'Z').
- A server may be repeatedly defined and re-defined when it is not connected. If you attempt to re-define a server that is currently connected a fatal error will result.
- By default, the same connection parameters will be used as when the OTHER File was loaded.
- The Connection Parameter overrides may be used to augment and replace the parameters used when connecting to this database. Existing parameters are only replaced on a parameter by parameter basis. Thus, the parameters that pre-exist and are not specified in the override parameter remain as they are.
- If this database is to be connected to by using the CONNECT_SERVER BIF, ALL the connection parameters will need to be specified in the Connection Parameter Overrides, otherwise ODBC will prompt for missing parameters. (Note: the following ODBC connection parameters are not required, and are ignored: DSN=, FILEDSN=, DRIVER=). Also, the Database Type Override must be specified, as the default Database Type is held in the OAM, which is not known when the CONNECT_SERVER BIF is used.
- When a Function or Component uses an OTHER File, if it has not been connected to a different server or database using CONNECT_FILE, the default database is used. The system checks if the database has been defined by this BIF. If it has, the Connection parameter Overrides are applied before an automatic connection is made.
- When a LANSAs File is to be connected to this database and AUTO_RRN is not used, it is mandatory that the RRN Path be specified.

- The database type has been provided for future use in supporting OTHER Files in Linux servers when the table definition is loaded into LANSAs using ODBC Oracle, but executes using native Oracle. If it is used to change from one ODBC database type to another a fatal error will occur when the OAM is executed. That is, the database type specified in this BIF must match the database type in the OAM, except for the special Oracle case.
- The details defined via this Built-In Function are not persistent. They are lost when the X_RUN command completes. You may choose to define your own set of SQL based tables to hold server details and actually read the table(s) to get values to be passed on to this Built-In Function.
- If the Connection parameter Overrides cause the user to be prompted for connection information (for example, a database login and password), the Connection parameter Overrides that were defined with this BIF may be updated with the actual connection string that was used to connect to the database. This feature is to save the user from being prompted for each new ODBC connection to this SSN (for example, SQL Server requires an update connection and multiple read connections), or where a DISCONNECT_SERVER is used, followed by a later CONNECT_SERVER. If you want to execute DISCONNECT_SERVER to force the user to be prompted again for connection details, you should call this BIF again to replace your original values.
- Please experiment with these facilities first and then design some sort of server architecture for your organization that has these characteristics:
 - It matches your organization's requirements.
 - It is quick and easy to change.
 - It is extensible.
- Do this before launching into any large-scale design or development project.

Example

The following example defines a database server called CHICAGO that connects to the database MYDATABASE and sets the user id and password to null so that ODBC prompts for the user id and password.

```
USE BUILTIN(DEFINE_DB_SERVER) WITH_ARGS(CHICAGO MYDATA
```

The connection will be made when the first file that is associated with MYDATABASE is used.

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application:

```
if (#retcode *ne OK)
  abort msgtxt('Failed to .....')
endif
```

Let the standard error handling built into every generated application, take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

9.38 DEFINE_OVERRIDE_FILE

⇒ **Note:** Built-In Function Rules.

Use this Built-In Function to override the Database Table Owner Name and/or the Database Table Name.

Typically this Built-In Function will be used when the library (schema name) associated with a file (table) is modified during the application installation. In this case the table is created in the database with the installation schema name but the supplied OAM has embedded in it the LANSAs Library of when the OAM was generated. To access the file when executing the application, the application must redirect the OAM to the appropriate LANSAs Library for the current installation using the DEFINE_FILE_OVERRIDE. Refer to [Why are File Overrides required?](#) and [Use of Define_Override_File with SuperServer and LANSAs Open.Net](#) for details.

The database overrides remain only during the current session.

The DEFINE_OVERRIDE_FILE's functionality is not available to the SELECT_SQL Free Format command.

For use with

LANSAs for i	No
Visual LANSAs for Windows	Yes
Visual LANSAs for Linux	No

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	File Library name If no File Library name is given, the BIF will remove all defined overrides.	1	10		
2	A	O	File name	1	10		

3	A	O	Override Database Table Owner name If no override database table owner and table name is given, the BIF will try to remove any previously defined override.	1	128		
4	A	O	Override Database Table name If no override database table owner and table name is given, the BIF will try to remove any previously defined override.	1	128		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Return Code: OK	2	2		

Why are File Overrides required?

When installing an application only one OAM will be installed which has embedded in it the LANSAs Library of when the OAM was generated. If an application file is installed into a library which is different to the library embedded in the OAM, the OAM must be redirected to the appropriate LANSAs Library at runtime. The LANSAs Library is mapped into the database equivalent, which has a different name in different database managers: it is variously called Schema, Owner and Collection.

A summary of the process:

1. The Function/Component is generated with the BUILT library and passes this to the common database execution layer.
2. The OAM is located. If this is in the *Partition Module Library* or *Partition File Library* then the library is no longer the BUILT library. It is an EXECUTE Library.

A file OAM refers to the library it is BUILT with. When overriding, you would specify this BUILT library in `OVERRIDE_TABLE_OWNER`.

3. The EXECUTE is looked up to see if there is an OVERRIDE. If the EXECUTE library is still the BUILT library then the file will be overridden with the new library.

If you need to override a file that has been installed in the *Partition File Library*, then the *Partition File Library* system variable needs to be assigned to a *Field* and this *Field* passed to the OVERRIDE_TABLE_OWNER so that the library can be overridden at runtime.

An application has files installed into the *Partition File Library*. Typically the *Partition File Library* is NOT the same name as the BUILT library. If it were the same, then OVERRIDE of BUILT would also override the *Partition File Library* files. This would not achieve the desired outcome, as the files should always access the same library.

Use of Define_Override_File with SuperServer and LANSA Open.Net

The DEFINE_OVERRIDE_FILE operates on the database local to the RDML that is executing. When database IO is performed, the OAM checks the file overrides on the machine/process where the OAM is executing. Therefore, an OAM executing on a server requires that the file overrides have been set up on the server by using CALL_SERVER_FUNCTION and calling the DEFINE_OVERRIDE_FILE appropriately. This is true for both SuperServer and LANSA Open .Net.

How to set up an override

1. A valid File Library name and optionally a File Name must be provided to indicate the file(s) that overrides should be applied to.
2. A valid Override Database Table Owner name and/or Override Database Table name must also be provided. The below table illustrates all valid cases:

Description	File Library Name	File Name	Override Database Table Owner Name	Override Database Table Name
Override all Files under a specific File Library name to a different Database Table Owner	X	*Default	X	
Override a specific File to a	X	X	X	

different Database Table Owner.				
Override a specific File to a different Database Table with the same Database Table Owner.	X	X	*Default	X
Override a specific File to a different Database Table.	X	X	X	X

How to remove an override of group of overrides

1. Execute DEFINE_OVERRIDE_FILE with no arguments values supplied to remove all overrides on database files.
2. Execute DEFINE_OVERRIDE_FILE with only the File Library Name argument to remove overrides from all files in a library.
3. Execute DEFINE_OVERRIDE_FILE with the File Library Name and File Name arguments to remove overrides from a specific file.

Description	File Library Name	File Name	Override Database Table Owner Name	Override Database Table Name
Remove all defined overrides				
Remove a File Library override	X			
Remove a specific File override	X	X		

Examples

Example 1: Override all Files under a specific File Library name to a

different Database Table Owner ABC

Use BIF(Define_Override_File) ('DC@DEMOLIB' *Default 'ABC') To_Get(#

Example 2: Override a specific File to a different Database Table Owner ABC

Use BIF(Define_Override_File) ('DC@DEMOLIB' 'PSLMST' 'ABC') To_Get

Example 3: Override a specific File to a different Database Table with the same Database Table Owner

Use BIF(Define_Override_File) ('DC@DEMOLIB' 'PSLMST' *Default 'XYZ'

Example 4: Override a specific File to a different Database Table

Use BIF(Define_Override_File) ('DC@DEMOLIB' 'PSLMST' 'ABC' 'XYZ') T

Example 5: Remove all defined overrides

Use BIF(Define_Override_File) To_Get(#retcode)

Example 6: Remove a File Library override

Use BIF(Define_Override_File) ('DC@DEMOLIB') To_Get(#retcode)

Example 7: Remove a specific File override

Use BIF(Define_Override_File) ('DC@DEMOLIB' 'PSLMST') To_Get(#retco

9.39 DEFINE_OS_400_SERVER

⇒ **Note:** Built-In Function Rules.

Defines details of an IBM i system that is to be used as a server to the current RDML function.

The definition details are not persistent and only exist while the LANSAs environment remains active.

The time taken to define a server is minimal.

You must use BIF DEFINE_ANY_SERVER for I/O commands to RDMLX files or to call an RDMLX function on the server.

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	SSN (Server Symbolic Name) This is the name that will be used in all future references to this server by this and other RDML functions.	1	10		
2	A	Req	LU Partner Name	1	20		
3	A	Req	Start commitment control on the server. Once this server is connected it will receive a "commit" or "rollback" request whenever the client issues an RDML level COMMIT or ROLLBACK command.	1	1		

			Y or 1 - Use commitment control other - do not use commitment control				
4	A	Opt	The server is DBCS capable. Y or 1 - DBCS capable server. Other - not DBCS capable server The default value is N.	1	1		
5	A	Opt	Divert LOCK_OBJECT requests to this server. If this option is used all subsequent LOCK_OBJECT requests will be diverted to this server. Multiple servers will receive the same LOCK_OBJECT request if multiple servers have this option enabled concurrently. In such cases a lock must be granted on all participating servers for the LOCK_OBJECT to complete normally. Where one server fails to grant the lock an UNLOCK_OBJECT request is made to all servers that have already granted the object lock. Note that *AUTONUM, *AUTOALP, and *DTASSLLLXXXXXXXXXX system variables are also retrieved from the server if locks are diverted to the server. Refer to *AUTONUM and *AUTOALP System Variables (Data Areas) and *DTASSLLLXXXXXXXXXX System Variables (Data Areas) in the <i>LANS Application Design Guide</i> . Y or 1 - Divert LOCK_OBJECT requests. Z - Divert LOCK_OBJECT requests and also divert authority checking requests to this server (only one server should be nominated as the diversion target for authority checking requests). R - Route lock requests AND authority requests AND repository data requests (if not found locally). Refer to the X_RUN	1	1		

			parameters in Using the X_RUN Command . Other- do not divert LOCK_OBJECT requests. The default value is N.				
6	A	Opt	Show "Please Wait" message while connecting. Y or 1 - Show wait message. Other - do not show message. Default value is Y.	1	1		
7	A	Opt	IBM i execution priority. Default value is 20. Specify other values as in the IBM i command CHGJOB parameter RUNPTY. User should be authorized to change to the nominated value.	1	2		
8	A	Opt	Client-to-Server conversion table name to be used. No library name can be specified. Defaults to ANSEBC1140.	1	10		
9	A	Opt	Server-to-Client conversion table name to be used. No library name can be specified. Defaults to EBC1140ANS.	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Server Defined ER - Server not defined and error message(s) issued.	2	2		

Technical Notes

- It is very strongly recommended that server definition logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to the server(s) that are being used.
- The LU partner name specified must be either:
 - The fully qualified Partner LU (Logical Unit) name that has been previously defined to the Communications Router.

This is usually formatted "Network Name.Control Point Name" and corresponds to the IBM i DSPNETA values of "Local Network ID" and "Local control point name".

Normally any Communications Router that allows 5250 emulation sessions to work will have these details already configured ready for use.

e.g. APPN.SYDASD25

- Or, if you have configured an "alias" name for the LU name it may be used in place of the fully qualified LU Name.

e.g. 5250PLU
- The translation tables specified are actually uploaded from the IBM i server during a connect. The initial connect phase needs to translate the initial connection details via a default translation table as the tables you specify have not yet been uploaded.

The initial connect details (sent to the server) include:

- The partition
- The language code
- The task identifier
- The client to server conversion table name
- The server to client conversion table name

Use only English alphabetic characters A through Z and the numbers 0 through 9 in the names of the objects listed above to avoid potential problems in this area.

- It is recommended that you use SSN values that are meaningful to end users (e.g. CHICAGO, BOSTON, CHARLIE1, etc.) as they may appear in

messages from time to time.

- SSN names must be unique and start with a character in the English language alphabet (uppercase A through Z).
- A server may be repeatedly defined and (re)defined when it is not connected. If you attempt to (re)define a server that is currently connected a fatal error will result.
- The details defined are not persistent. They are lost when the X_RUN command completes. You may choose to define your own set of SQL based tables to hold server details and actually read the table(s) to get values to be passed on to this Built-In Function.
- Please experiment with these facilities first and then design some server architecture for your organization that has these characteristics:
 - It matches your organization's requirements.
 - It is quick and easy to change.
 - It is extendible.

Do this before launching into any large scale design or development project.

- The client's date format will be automatically passed to the server. If the date formats are different (e.g. MDY vs DMY), the server will automatically return data in the client's format.

The client's date format can be changed from the default by specifying the x_run parameter DATF=. Please refer to [Standard X_RUN Parameters](#) for more information about this parameter.

Note that if the client and server date formats are different, Date format validation rules specifying exact formats (e.g. DDMMYY) will fail (as the data may be returned as MMDDYY). Date format SYSFMT is recommended where clients need to use different date formats (e.g. USA and UK clients).

Notes on Commitment Control

- If Start Commitment Control is Y, LANSAs will automatically start and end commitment control. See [User Exit F@BGNCMT - Start Commitment Control](#) and [User Exit F@ENDCMT - End Commitment Control](#) in the *LANSAs for i User Guide* for details.
- When the server has been indicated as having commitment control started, it

will effect all subsequent COMMIT and ROLLBACK commands issued. When a COMMIT or ROLLBACK command is issued the routine involved loops through all currently connected servers.

To each one that has commitment control active, it issues a "commit" or "rollback" request and then waits for the server to respond before proceeding. This is done after a commit/rollback has been issued correctly to the local/client database management system.

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application:

```
if (#retcode *ne OK)

    abort msgtxt('Failed to .....')

endif
```

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

DBCS Considerations

When the server is indicated as being DBCS capable an additional translation table must be locatable on the client PC.

This table must be named X_CT<language code>.DAT and must be located in the X_LANSA\EXECUTE directory.

A version of this table named X_CTJPN.DAT for <language code> "JPN" (Japanese) is available to all Japanese customers.

This translation file contains the following notes:

- This table is shipped in an "as is" condition to support customer tailoring. No warranty is expressed or implied. It is the customer's responsibility to maintain and verify this table.
- This table is loaded by the Visual LANSA DEFINE_OS_400_SERVER function when it has been indicated that the server has DBCS capability.

- The name of the table loaded is derived from "x_ct" combined with the current language code (e.g. jpn) and the suffix ".dat". Thus if language code jpn was being used, the table name would have to be "x_ctjpn.dat". For language code tchi the table name would be "x_cttchi.dat".
- The table must reside in the <drive>:\x_lansa\execute directory where <drive> is whatever local or server disk drive onto which Visual LANSA has been installed.
- This conversion table is only used for the double byte parts of any string. Single byte parts of any string at all (DBCS allowed or not) are always converted by using the IBM i single byte tables specified in the DEFINE_OS_400_SERVER Built-In Function call.
- This means that a field containing mixed double and single byte characters is partially converted by this table and partially by the single byte conversion table that is used by both DBCS and non-DBCS conversions.
- DBCS conversion of data within an individual field only occurs when the field is indicated as being DBCS capable (e.g. dictionary keyboard attributes j, e, o, etc.) and when the DEFINE_OS_400_SERVER Built-In Function has indicated that the server is DBCS capable. If both these conditions are not met the entire field is converted as a single byte string by the single byte conversion tables previously mentioned.
- An * in column 1 indicates a comment line.
- All values are specified in hexadecimal format.
- Separate values with a single comma (,) only.

9.40 DEFINE_OTHER_SERVER

⇒ **Note:** Built-In Function Rules.

Defines details of a non-IBM i (i.e. other) system that is to be used as a server to the current RDML function.

The definition details are not persistent and only exist while the LANSAs environment remains active. The time taken to define a server is minimal.

To use this BIF you must set x_run parameter CDLL to LCOMGR32.DLL and x_run parameter CMTH must be C or T.

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	SSN (Server Symbolic Name) This is the name that will be used in all future references to this server by this and other RDML functions.	1	10		
2	A	Req	Server Network Name	1	20		
3	A	Opt	Divert LOCK_OBJECT requests to this server. If this option is used all subsequent LOCK_OBJECT requests will be diverted to this server. Multiple servers will receive the same LOCK_OBJECT request if multiple servers have this option enabled concurrently.	1	1		

			<p>In such cases a lock must be granted on all participating servers for the LOCK_OBJECT to complete normally. Where one server fails to grant the lock an UNLOCK_OBJECT request is made to all servers that have already granted the object lock. Note that *AUTONUM, *AUTOALP, and *DTASSLLLXXXXXXXXXX system variables are also retrieved from the server if locks are diverted to the server. Refer to *AUTONUM and *AUTOALP System Variables (Data Areas) and *DTASSLLLXXXXXXXXXX System Variables (Data Areas) in the <i>LANSAs Application Design Guide</i>.</p> <p>Y or 1 - Divert LOCK_OBJECT requests. Z - Route lock requests AND route authority requests. R - Route lock requests AND authority requests AND repository data requests (if not found locally). Refer to the X_RUN parameters in Using the X_RUN Command. Other - do not divert requests. The default value is N.</p>				
4	A	Opt	<p>Show "Please Wait" message while connecting. Y or 1 - Show wait message. other - do not show message. Default value is Y.</p>	1	1		
5	A	Opt	X_RUN exceptional arguments.	1	256		
6	A	Opt	<p>Server dependent exceptional arguments. Not currently implemented. Do not use this argument.</p>	1	256		
7	A	Opt	Reserved for Future Expansion. Not currently implemented.	1	256		

			Do not use this argument.				
8	A	Opt	Reserved for Future Expansion. Not currently implemented Do not use this argument.	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Server Defined ER - Server not defined and error message(s) issued.	2	2		

Technical Notes

- The Server Network Name you specify when invoking this Built-In Function should be identical to the Partner LU Name under which the server was (or will be) enrolled within the LANSAs Communications Administrator.
- It is very strongly recommended that your server definition and connection logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to the server(s) that are being used.
- It is recommended that you use SSN values that are meaningful to end users (e.g: CHICAGO, BOSTON, CHARLIE1, etc.) as they may appear in messages from time to time.
- SSN names must be unique and start with a character in the English language alphabet (uppercase A through Z).
- A server may be repeatedly defined and (re)defined when it is not connected. If you attempt to (re)define a server that is currently connected a fatal error will result.
- The X_RUN exceptional argument may be used to override the parameters

used on the X_RUN command started on the server system.

By default, the following client X_RUN parameter values are passed to (and inherited by) the X_RUN command started on the server system: CMTH=, CDLL=, DATF=, DATS=, DBCF=, DBCL=, DBLK=, DBTC=, DBUS=, DEVE=, FXQF=, FXQM=, HSKC=, INIT=, ITHP=, ITRC=, ITRL=, ITRM=, ITRO=, LANG=, PART=, PRTR=, PSPW=, PSTC=, PSWD=, TASK=, TERM=, USER=, XAFP= and XCMD. The following client X_RUN parameter values are only inherited by the server if the client and the server are using the same operating system: DBID=, DBII=, DBIT=, DBUT=, ODBI= and WPEN (and related Windows Printing Extension [parameters](#)).

All other X_RUN parameter values on the server system are defaulted (on the server system) in the usual manner (ie: from a profile file, from system environment settings, etc). Refer to the definition of the X_RUN command for details of all parameter values and the methods by which they can be specified and defaulted.

- You may override any server X_RUN parameter (via the X_RUN exceptional argument value) except for CDLL=, CMTH=, DATF=, DATS=, DEBUG=, DEVE=, LANG=, MODE=, PART=, PROC=, PSPW=, USER= and XAFP=. These X_RUN arguments are unconditionally inherited from the client system. However, some of these parameters may be altered by calling [SET_SESSION_VALUE](#) before invoking CONNECT_SERVER.

Override parameters may be given a specific value, or the special value *SERVER, which indicates that the server default should be used. As an example, a Windows client using DBII=*NONE might connect to a Windows Server running Oracle. By default, Windows uses the database type MSSQLS (SQL Server), so DBUT needs to be overridden. The X_RUN exceptional argument value could be set to either DBUT=ODBCORACLE or DBUT=*SERVER.

- The details defined via this Built-In Function are not persistent. They are lost when the X_RUN command completes. You may choose to define your own set of SQL based tables to hold server details and actually read the table(s) to get values to be passed on to this Built-In Function.
- Please experiment with these facilities first and then design some sort of server architecture for your organization that has these characteristics:
 - It matches your organization's requirements.

- It is quick and easy to change.
- It is extensible.

Do this before launching into any large-scale design or development project.

- The client's date format will be automatically passed to the server. If the date formats are different (e.g. MDY vs DMY), the server will automatically return data in the client's format.

The client's date format can be changed from the default by specifying the `x_run` parameter `DATF=`. Please refer to [Standard X_RUN Parameters](#) for more information about this parameter.

Note that if the client and server date formats are different, Date format validation rules specifying exact formats (e.g. DDMMYY) will fail (as the data may be returned as MMDDYY). Date format SYSFMT is recommended where clients need to use different date formats (e.g. USA and UK clients).

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application:

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
endif
```

Let the standard error handling built into every generated application, take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

9.41 DEFINE_SPACE_CELL

⇒ **Note:** Built-In Function Rules.

Defines a cell (or column) within the nominated space object. Refer also to the other **SPACE** Built-In Functions.

For use with

LANSA for i	YES	Only available with RDMLX
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Name of the space into which the cell should be defined.	1	256		
2	A	R	Name of a field whose definition is to be used as a prototype for the definition of this space cell. Note that this is the name of the field, not the field itself. Thus you should specify an employee number field as EMPNO rather than #EMPNO (which means that the content of field #EMPNO contains the field name).	1	10		
3	A	O	Cell attributes. Multiple cell attributes should be separated by a single space. The list of valid attributes (which may be in any case) includes: KEY: specifies that this cell is a key to the	1	256		

		<p>space contents. Space objects should always have at least one key cell.</p> <p>DESCEND: specifies that descending sequence should be applied to this cell. This attribute is only valid when used with the KEY attribute, otherwise it is ignored.</p> <p>NOCASE: specifies that the case of the key is ignored and all key comparisons are done in lowercase. This attribute is only valid when used with alphanumeric KEY cells, otherwise it is ignored.</p>				
--	--	--	--	--	--	--

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	<p>Standard Return Code</p> <p>"OK" = Cell defined.</p> <p>"ER" = Cell definition attempt failed.</p> <p>Messages issued will indicate more about the cause of the failure.</p>	2	2		

Technical Notes

The specified space object must be defined before any attempt is made to define cells within it (see the CREATE_SPACE Built-In Function)

The currently active invocation stack is searched backwards until the first instance of a field with the specified name is found.

If an instance of the field name is found it is then instantaneously used as a definitional prototype for the space cell in terms of type, length, etc.

If an instance of the field name cannot be found a fatal error message is issued and the application terminates.

It is recommended that all KEY cells be defined as the first cells in a space object.

The order that a key cell is defined implies their order within the aggregate key to the cell row.

The order that any cell is defined implies the order that they will be mapped into and out of other space commands such as INSERT_IN_SPACE and FETCH_IN_SPACE.

If the data in the space requires more than one level of key to uniquely identify an individual entry, then more than one cell MUST be defined as the key.

Examples

Example 1

This example defines a space whose name is the current components name suffixed by ".emp" and then defines 3 cells within it whose type and length are based on the definitions of fields EMPNO, GIVENAME and SURNAME respectively. The first cell the key to the space:

```
Define #SpaceName *char 20
Use TConcat (*component '.EMP') (#SpaceName)
Use Create_Space (#SpaceName)
Use Define_Space_Cell (#SpaceName EmpNo Key)
Use Define_Space_Cell (#SpaceName GiveName)
```

```
Use Define_Space_Cell (#SpaceName SurName);
```

Example 2

The section file (SECTAB) in the LANSA demonstration system requires that two levels of key be specified to identify any given record. The space must be defined in a similar format. Failure to do so will cause unpredictable results when attempting to retrieve data from the space using SELECT_IN_SPACE and SELECT_NEXT_IN_SPACE.

```
Define field(#SpaceName) type(*char) length(20)
Use TConcat (*component '.EMP') #SpaceName)
Use Create_Space (#SpaceName)
Use Define_Space_Cell (#SpaceName Deptment Key)
Use Define_Space_Cell (#SpaceName Section Key)
Use Define_Space_Cell (#SpaceName Secdesc)
```

9.42 DELETE_CHECKS

⇒ **Note:** Built-In Function Rules.

Deletes standard DICTIONARY or FILE level validation checks from a nominated field for subsequent replacement by PUT_XXXXXXX validation check Built-In Functions.

When deleting FILE level validation checks from a field, the file involved must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

Normal authority and task tracking rules apply to the use of this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation checks that are to be deleted. D = Dictionary level F = File level	1	1		
2	A	Req	Name of field in dictionary or file from which validation rules are to be deleted.	1	10		
3	*N	Req	Sequence number to control deletion. Only checks with a sequence number greater than	1	3	0	0

			or equal to this value are deleted. If this argument is not specified, a value of zero (0) is assumed, so all checks will match this control value.				
4	*A	Req	Generic description of check used to control deletion. Only checks which have a description generically matching this value will be deleted. If this value is not specified, a default value of blanks is assumed, so all checks will match this control value.	1	30		

* The deletion control sequence number and description are related by an "AND" relationship. So if you pass values of 500 and IEW, only checks that have a sequence number greater than or equal to 500 and a description that starts with IEW will be deleted.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected NR = no records found eligible for deletion In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

Example

A user wants to delete validation checks for a specific field, without going through the LANSAs options provided on the *Field Control Menu* that enables

the user to delete validation checks.

```
***** Define arguments and lists
DEFINE FIELD(#LEVEL) TYPE(*CHAR) LENGTH(1) LABEL('Level')
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(10) LABEL('Field')
DEFINE FIELD(#SEQNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
    LABEL('Sequence #')
DEFINE FIELD(#DESCR) TYPE(*CHAR) LENGTH(30) LABEL('Descrip
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2) LABEL('Return
GROUP_BY NAME(#VALCHK) FIELDS((#LEVEL) (#FIELD) (#SEQNUM
***** Request Validation check details
BEGIN_LOOP
REQUEST FIELDS(#VALCHK)
***** Execute built-in-function - DELETE_CHECKS
USE BUILTIN(DELETE_CHECKS) WITH_ARGS(#LEVEL #FIELD #SI
    #DESCR) TO_GET(#RETCOD)
***** Deletion of validation checks was successful
IF COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Deletion of validation check(s) was successful')
***** Deletion of validation checks failed
ELSE
IF COND('#RETCOD *EQ "ER"')
MESSAGE MSGTXT('Deletion of validation check(s) failed')
***** No records found eligible for deletion
ELSE
IF COND('#RETCOD *EQ "NR"')
MESSAGE MSGTXT('No Records found eligible for deletion')
ENDIF
ENDIF
ENDIF
END_LOOP
```

9.43 DELETE_FUNCTION

⇒ **Note:** Built-In Function Rules.

Deletes all details of the function currently being edited and ends the edit session against the function. An edit session is commenced by using the Built-In Function START_FUNCTION_EDIT.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions. This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No argument values.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = function deleted and edit session was ended ER = fatal error detected	2	2		

9.44 DELETE_IN_SPACE

⇒ **Note:** Built-In Function Rules.

Deletes all cell rows that match the key values supplied.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name	1	256		
2-20	w	O	Fields specifying the key values to be used to locate the cell rows to be deleted.	1	Unlimited		Unlimited

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "OK" = A cell row was found and the cell values have been returned. "NR" = Now cell row could be found with a key matching the key values supplied. "ER" = Select attempt failed. Messages	2	2		

		issued will indicate more about the cause of the failure.				
--	--	---	--	--	--	--

Technical Notes

You can specify less key values than are defined in the space. All matching cell rows will be deleted. This means that partial keys operations can be performed.

The operation `USE DELETE_IN_SPACE (CustomerList)` will delete all cell rows from the space named CustomerLists.

If you specify more key values than are defined as key cells for the space then the additional values will be ignored and have no effect on the outcome of the delete operation.

If a key value longer than 256 bytes is specified, a fatal error will occur.

9.45 DELETE_PROCESS

⇒ **Note:** Built-In Function Rules.

Submits a job to delete a process and all of its functions.

Argument values are exactly as the information input on the "Delete a Process" screen described in [Submitting the Job to Delete a Process Definition](#) in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition. LANSA for i submits a job to batch as a separate task.
Visual LANSA for Windows	YES	Visual LANSA initiates the delete process and does not return control until the delete is complete.
Visual LANSA for Linux	NO	

Arguments for Visual LANSA

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Process name	1	10		
2	A	Opt	Name of job Ignored	1	10		
3	A	Opt	Name of job description Ignored	1	21		
4	A	Opt	Name of job queue Ignored	1	21		

5	A	Opt	Name of output queue Ignored	1	21		
---	---	-----	---------------------------------	---	----	--	--

Arguments for LANSA for i

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Process name	1	10		
2	A	Opt	Name of batch job Default: Process name	1	10		
3	A	Opt	Name of job description Default: the job description from the requesting job's attributes.	1	21		
4	A	Opt	Name of job queue Default: the job queue from the requesting job's attributes.	1	21		
5	A	Opt	Name of output queue Default: the output queue from the requesting job's attributes.	1	21		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = successful submission ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued	2	2		

			automatically.				
--	--	--	----------------	--	--	--	--

Example

A user wants to control the deletion of processes using their own version of the "Delete a Process" facility.

```
***** Define arguments
DEFINE  FIELD(#PROCES) TYPE(*CHAR) LENGTH(10)
DEFINE  FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
***** Request Process
BEGIN_LOOP
REQUEST  FIELDS(#PROCES)
***** Execute built-in-function - DELETE_PROCESS
USE      BUILTIN(DELETE_PROCESS) WITH_ARGS(#PROCES) TO_GE'

***** Check if submission was successful
IF      COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Delete Process submitted successfully')
CHANGE  FIELD(#PROCES) TO(*BLANK)
ELSE
MESSAGE MSGTXT('Delete Process submit failed with errors,
refer to additional messages')
ENDIF
END_LOOP
```

9.46 DELETE_SAVED_LIST

⇒ **Note:** Built-In Function Rules.

Deletes a previously saved permanent or temporary working list.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of previously saved list to be deleted.	1	10		

Return Values

No return values.

Technical Notes

- This Built-In Function is designed to delete a permanent or temporary working list.
- It is good practice to specifically delete temporary lists.
- File DC@F80 in the LANSA data library is used to store saved list details. This file should be considered in your backup and restore procedures.
- Deleted record space in file DC@F80 is reorganized and removed during a normal LANSA internal database reorganization. This reorganization also deletes any temporary lists that have exceeded their retention period.

- You can reorganize file DC@F80 to free deleted record space at any time by using the IBM i RGZPFM (Reorganize Physical File Member) command. Use DC@F80V1 as the sequencing logical view.
- The backup and recovery of data area DC@A08 and database file DC@F80 (and its logical views DC@F80V1 and DC@F80V2) is **your** responsibility.
- Movement of DC@F80 or saved lists between machines or between environments is your responsibility.

Example

A list has been saved. It contains a list of contacts that need to have their information updated in the database. This user is in a telemarketing role. The user looks at a record, calls the contact and ensures the database is up to date. If a change is necessary the user changes the information and updates the database.

The list has come from a job that has checked the date of update of each contact. If the update date is more than NN days the record is put into a list. The list is then saved.

When the list has been processed the saved list is deleted.

```

DEF_LIST  NAME(#RSTLST) FIELDS((#CTTCDE) (#CTTDES))
          TYPE(*WORKING)
GROUP_BY  NAME(#DETAIL) FIELDS((#CTTCDE) (#CTTDES) (#CTTN
          (#CTTAD1) (#CTTAD2) (#CTTAD3) (#CTTPHO) (#CTTFAX)
          (#CTTUPD))
***** Clear the list
CLR_LIST  NAMED(#RSTLST)
***** Restore the list
USE       BUILTIN(RESTORE_SAVED_LIST) WITH_ARGS('CONTACTS')
          TO_GET(#RSTLST)
***** Process the list
SELECTLIST NAMED(#RSTLST)
FETCH     FIELDS(#DETAIL) FROM_FILE(CONTACTS) WITH_KEY(#CI
DISPLAY   FIELDS(#DETAIL) CHANGE_KEY(*YES)
***** If contact information has changed
IF_MODE   IS(*CHANGE)
CHANGE    FIELD(#CTTUPD) TO(*YYMMDD)
UPDATE    FIELDS(#DETAIL) IN_FILE(CONTACTS)
ENDIF
ENDSELECT

```

******* Delete the list**

USE BUILTIN(DELETE_SAVED_LIST) WITH_ARGS(#LSTNME)

9.47 DELETE_TRIGGERS

⇒ **Note:** Built-In Function Rules.

Deletes standard DICTIONARY or FILE level triggers from a nominated field for subsequent replacement by the PUT_TRIGGER Built-In Function.

When deleting FILE level triggers from a field, the file involved must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

Normal authority and task tracking rules apply to the use of this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of triggers that are to be deleted. D = Dictionary level F = File level	1	1		
2	A	Req	Name of field in dictionary or file from which triggers are to be deleted.	1	10		
3	*N	Req	Sequence number to control deletion. Only triggers with a sequence number greater than or equal to this value are deleted. If this argument is not specified, a value of zero (0)	1	3	0	0

			is assumed, so all triggers will match this control value.				
4 *	A	Req	Generic description of trigger used to control deletion. Only checks which have a description generically matching this value will be deleted. If this value is not specified, a default value of blanks is assumed, so all checks will match this control value.	1	30		

* The deletion control sequence number and description are related by an "AND" relationship. So if you pass values of 500 and 'IEW', only triggers that have a sequence number greater than or equal to 500 and a description that starts with 'IEW' will be deleted.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected NR = no records found eligible for deletion In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

9.48 DELETE_WEB_COMPONENT

⇒ **Note:** Built-In Function Rules.

Programmatically delete a Web Component. Refer to the *LANSA for the Web Guide* for more information about Web Components.

All versions (including backups) of the page text will be deleted.

This Built-In Function only supports page, script, text, and visual components. An error will be issued if an attempt is made to delete a banner, weblink, or file component.

When deleting a component, if a language is specified, only the X02 records for that language will be deleted. If *ALL is used for the language, X02 records for all languages are deleted, as well as the X03 and X01 header records.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Web Component	1	20		
2	A	Req	Mode. Possible values are: I: Input O: Output N: Not applicable.	1	1		
3	A	Opt	Language Default is all languages (*ALL) For non-multilingual partitions this should be 'NAT'	4	4		

		*DFT - partition default language.				
--	--	------------------------------------	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Possible values are: OK: Delete completed normally. NR: No component found to delete. ER: Delete encountered an error.	1	2		

9.49 DESTROY_SPACE

⇒ **Note:** Built-In Function Rules.

Destroys the space object with the specified name.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name. Special space name *ALL may be used to destroy all spaces within the current operating system process.	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "OK" = Space destroyed or does not exist. "ER" = Destruction attempt failed. Messages issued will indicate more about the cause of the failure.	2	2		

Technical Notes

Once a space object has been destroyed it can no longer be used.

9.50 DISCONNECT_FILE

⇒ **Note:** Built-In Function Rules.

Disconnects a file previously connected to a server.

Warning:

- Using this Built-In Function during pending I/O operations to the file (e.g. in the middle of a SELECT loop) will cause unpredictable results.
- A request to disconnect a file that is not currently connected will be ignored. No error will result.

For use with

LANSA for i	YES	Only available with RDMLX
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Physical File\Table Name The name may be specified as a generic name. The * symbol is used as the generic delimiter.	1	10		
2	A	Req	SSN of defined server.	1	10		

Return Values

No return values.

Technical Notes

- When a generic name is specified it must exactly match a generic name previously defined by using the CONNECT_FILE Built-In Function.
- For example connecting file name A* and then disconnecting file name AB* does not mean that all files starting with A (except those starting with AB) are currently connected. The disconnect of AB* will not find a match in the connected files list and be ignored. All files that start with A will continue to be connected.
- Disconnecting a file while it is in use (e.g: in the middle of a SELECT loop when the file being selected is not connected to a server or connected to another server) will cause application failure and/or unpredictable results.
- It is very strongly recommended that all "disconnect" logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to the server(s) that are being used.
- Do not attempt to disconnect a blank file name.
- When using generic file names (e.g. LM*, GL*, *) be extremely careful not to overlap any generic names. Failure to observe this rule will cause unpredictable results. This rule means that name "*" (any name) can only be used by itself, as any other file name disconnected before or after the "*" will overlap with it.
- There is no real need to disconnect file(s). As the X_RUN command is terminating it will automatically disconnect any connected files.
- Message information routed from the server machine (in any form) arrives in a text format. It is displayed and accessible to RDML functions in the normal manner (e.g. GET_MESSAGE) as pure text. The message identifier and message file name details are not available for messages that have been routed from a server. You should not design client applications that rely on reading specific message identifiers from the applications message queue.

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application.

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
```

endif

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

9.51 DISCONNECT_SERVER

⇒ **Note:** Built-In Function Rules.

Disconnects the current function from a previously connected server.

A request to disconnect a server that is already disconnected will be ignored. No error will result.

For use with

LANSA for i	YES	Only available with RDMLX
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	SSN of a defined server	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. OK - Disconnection Completed. ER - Error during Disconnection	2	2		

Technical Notes

- Disconnecting a server while it is in use (e.g. in the middle of a SELECT loop when the file being selected is connected to the server) will cause application failures.
- It is very strongly recommended that all "disconnect" logic is coded in one and only one function, rather than scattered and repeated through many RDML functions. This approach will isolate your application from future changes to the server(s) that are being used.
- There is no real need to disconnect server(s). As the X_RUN command is terminating it will automatically disconnect any connected servers.
- When executing against a database server, the actual connections to the database are dropped, but if more IO is performed against the database the connections will be re-established automatically.

A Note on Error Handling

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application.

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
endif
```

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

9.52 DLL

⇒ **Note:** [Built-In Function Rules](#).

Allows processing within a standard DLL (Dynamic Link Library) object to be invoked from a LANSAs component or function.

This is a generic interface to operating system DLL entry points. It may not be the most appropriate interface for specialized requirements. For specialized requirements please investigate creating your own Built-In Function. Refer to [Creating Your Own Built-In Functions](#) in the *LANSAs Application Design Guide* for more details of this facility.

For use with

LANSAs for i	NO	
Visual LANSAs for Windows	YES	Only execute in Visual LANSAs applications run on MS Windows platforms.
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Alias name of DLL to be invoked.	1	32		
2	A	Req	Alias name of function (i.e. entry point) to be invoked.	1	32		
3-20	X	Opt	User defined real or logical arguments to be passed to the DLL entry points.	1	Unlimited		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code "OK" = Completed okay. "ER" = Error detected.	2	2		
2	A	Opt	Error message text	1	256		
3- 20	X	Opt	User defined real or logical return values from the DLL entry point.	1	Unlimited		

Technical Notes

- It is obvious, but it needs to be said. By using this Built-In Function you are introducing an operating system dependency into your function.

You, the application builder, are totally responsible for doing this, and there is no guarantee, expressed or implied, that anything you do via this Built-In Function is in any way portable across different operating systems (or even versions of the same operating system).

- The DLL alias name and entry point alias name must be alphanumeric literals as the values that they specify must be determinable at compile time.
- Before a DLL can be used it must be defined in file X_USEDLL.DAT in the X_LANSA\SOURCE directory. This file provides details of the DLL and its entry points and maps alias names to real names. The alias and real name facility has been provided to:
 - Provide a level of isolation between the RDML function and the real DLL. For example, you can change the name of the real DLL being used without having to change code in your RDML functions (but you will have to recompile them as the real name is referenced in the generated C code).
 - To allow for RDML code that is easier to read. For example, the Crystal Reports DLL is called "CRPE", but when given an alias name of "CrystalReports" the resulting RDML code is much easier to understand. The same logic applies to entry point names.

The X_USEDLL.DAT Definition File

The X_USEDLL.DAT definition file defines the characteristics of all DLLs that can be used by the DLL Built-In Function. The following should be read and understood before attempting to create or alter this file or use this Built-In Function:

- It is an optional file. When present it must reside in the X_LANSA\SOURCE directory.
- It is only required on, and used by, development environments during code generation. Thus changes to the content of the file may require the (re)generation and (re)compilation of RDML functions that use DLLs affected by the changes.
- There is no need for this file to exist in execution environments.
- No DLL can have more than 256 entry points defined for it.
- This file is a simple keyword definition style file that can be edited with most source editors. The file is laid out as a series of keywords (which must begin in column 1 of the record, and only one keyword per line may be specified):

Keyword	Meaning
*	Line is a comment
DLLNAM=	Name of the real (ie: operating system level) DLL file without any form of suffix.
DLLALS=	The alias name for this DLL that is to be used in RDML functions. Up to 32 characters may be specified. This name must be unique within the definition file and is not case sensitive. The DLLNAM= must precede DLLALS=.
ENTNAM=	Name of the real (ie: operating system level) DLL entry point. This name is case sensitive in most operating systems. This name should be unique within the current DLL definition.
ENTALS=	The alias name for a DLL entry point that is to be used in RDML functions Up to 32 characters may be specified. This name must be unique within the current DLL and is not case sensitive. The ENTNAM= must precede ENTALS=.
ENTSKL=	Specifies the name of the skeleton file (including the suffix) that defines the required C code to execute the DLL entry point. This

code is used as a template to generate C code to invoke the DLL entry point. Please refer to shipped samples / examples for fully commented examples of how to produce such a template. All template files must reside in the X_LANSA/SOURCE directory. The ENTNAM= must precede ENTSKL=.

As an example of the content of this file, consider the following:

```
* -----  
* Example of defining Crystal Report DLL entry points  
* -----  
DLLNAM=CRPE  
DLLALS=CrystalReports  
*  
ENTNAM=PEOpenEngine  
ENTALS=OpenEngine  
ENTSKL=CRPE0001.S  
*  
ENTNAM=PECloseEngine  
ENTALS=CloseEngine  
ENTSKL=CRPE0002.S  
*  
ENTNAM=PEPrintReport  
ENTALS=PrintReport  
ENTSKL=CRPE0003.S  
*  
ENTNAM=PEGetVersion  
ENTALS=GetVersion  
ENTSKL=CRPE0004.S
```

Here you can see the definitions for the Crystal Reports DLL named CRPE. It has four entry point aliases called OpenEngine, CloseEngine, PrintReport and GetVersion. The skeleton / template code for invoking this DLL and the entry points is to be found in the ...\X_LANSA\SOURCE directory in files CRPE0001.S, CRPE0002.S, CRPE0003.S and CRPE0004.S respectively. The skeleton code defines how the DLL is loaded, what the real entry point names are, how Built-In Function arguments are mapped to entry point arguments, how

return codes and return values are handled, etc.

When setting up your own skeleton files / templates please follow these guidelines:

- Copy a shipped example and use it as a starting point.
- Always place the file in the\X_LANSA\SOURCE directory and make a backup copy.
- Do not use the file suffix .S, which is reserved so the Visual LANSA install/upgrade procedures can identify its own, shipped skeletons and replace them. If you use suffix .S then your files will be deleted the next time that you install/upgrade.

The DLL may need to be copied into the X_LANSA/EXECUTE directory for the application to be executed properly. For example, CRPE32.DLL has to be copied into the X_LANSA/EXECUTE directory.

At your development site, you need to decide on a convention for mapping this Built-In Function's arguments to the actual DLL entry point arguments. The convention you adopt is implemented in the skeleton/template code associated with each DLL entry point.

The skeleton template file supplied was used because it supports:

- Variable argument lists and default values for arguments that the caller has not passed.
- Variable and optional return values.
- The ability to add new arguments or return values at any time, without the need to modify or even recompile any existing caller of the DLL entry point.
- The ability to handle all calling conventions and pointer conversion requirements and, cleanly support multiple operating systems via one simple interface.
- The ability to hide the complexity of DLL loading and entry point resolution from the RDML level programming interface.
- Performance appropriate for the high level of programming used in the LANSA RDML language.

For example, the Crystal Reports entry points use the following parameter and return value conventions in their shipped format:

DLL Alias	Entry Point	Alias	Arguments And Return Values
CrystalReports	OpenEngine	Ret 1	Is the OK or ER return code and it is

			optional.
CrystalReports	CloseEngine	Ret 1	Is the OK or ER return code and it is optional.
CrystalReports	PrintReport	Arg 3	Is the report name and it is required. If not specified it will default to a null string causing a "not found" error.
		Arg 4	Is the print report option and it is optional. It should be Y or N. It defaults to Y.
		Arg 5	Is the show in window option and it is optional. It should be Y or N. It defaults to N.
		Arg 6	Is the window title and it is optional. It defaults to a null string.
		Ret 1	Is the OK or ER return code.
CrystalReports	GetVersion	Ret 1	Is the OK or ER return code and it is optional.
		Ret 2	Is the major version number and it is optional.
		Ret 3	Is the minor version number, and is optional.

9.53 DLT_FIELD

⇒ **Note:** Built-In Function Rules.

Deletes a field definition from the LANSAs Repository and any LANSAs for the Web visual components associated with the field.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of field to be deleted from Repository	1	10		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code. OK = field details returned ER = field not accessible. In case of "ER" return code error message(s) are issued automatically.	2	2		

9.54 DLT_FILE

⇒ **Note:** Built-In Function Rules.

Submits a job to delete a file and its associated logical files and I/O module.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition. On the IBM i , this Built-In Function submits a job to perform the delete operation.
Visual LANSA for Windows	YES	This Built-In Function does not submit a job. It deletes the file and then returns control.
Visual LANSA for Linux	NO	

Arguments for Visual LANSA

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	File name	1	10		
2	A	Req	Library name. In Visual LANSA blanks are valid for backward compatibility.	1	10		
3	A	Opt	Name of job. Ignored	1	10		
4	A	Opt	Name of job description. Ignored	1	21		
5	A	Opt	Name of job queue. Ignored	1	21		

6	A	Opt	Name of output queue. Ignored	1	21		
---	---	-----	----------------------------------	---	----	--	--

Arguments for LANSAs for i

For further information, refer to *Delete a file from the System* described in [Submitting the Job to Delete a File Definition](#) in the *LANSAs for i User Guide*.

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	File name	1	10		
2	A	Req	Library name	1	10		
3	A	Opt	Name of job Default: File name	1	10		
4	A	Opt	Name of job description Default: the job description from the requesting job's attributes.	1	21		
5	A	Opt	Name of job queue Default: the job queue from the requesting job's attributes.	1	21		
6	A	Opt	Name of output queue Default: the output queue from the requesting job's attributes.	1	21		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = successful submission ER = argument details are invalid or an authority problem has occurred.	2	2		

		In case of "ER" return code error message(s) are issued automatically.				
--	--	--	--	--	--	--

Example

A user wants to control the deletion of files and associated logical views and I/O module using their own version of the "Delete a file from the System" facility.

```
***** Define arguments and lists
DEFINE    FIELD(#FILNAM) TYPE(*CHAR) LENGTH(10)
DEFINE    FIELD(#LIBNAM) TYPE(*CHAR) LENGTH(10)
DEFINE    FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
BEGIN_LOOP
***** Request File and library name
REQUEST   FIELDS(#FILNAM #LIBNAM)
***** Execute built-in-function - DLT_FILE
USE       BUILTIN(DLT_FILE) WITH_ARGS(#FILNAM #LIBNAM)
          TO_GET(#RETCOD)
***** Check if submission was successful
IF        COND('#RETCOD *EQ "OK"')
MESSAGE   MSGTXT('Delete of file submitted successfully')
CHANGE    FIELD(#FILNAM) TO(*BLANK)
ELSE
MESSAGE   MSGTXT('Delete submit failed with errors,
                refer to additional messages')
ENDIF
END_LOOP
```

9.55 DLT_PROCESS_ATTACH

⇒ **Note:** Built-In Function Rules.

Deletes all attached processes and/or functions from the definition of the process definition currently being edited by the START_PROCESS_EDIT Built-In Function.

Information passed into this Built-In Function is subjected to the same editing and validation rules as the equivalent online facility provided in a full LANSAs development environment.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Delete Test Sequence Number. All attached processes / functions with a sequence number greater than or equal to this value are to be deleted.	1	3	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.56 DOM_ADD_FIELD

⇒ **Note:** Built-In Function Rules.

Adds a field to an open data note using the field name. The field type must also be specified.

The field types supported are the simple data types TYPE_TEXT, TYPE_NUMBER, TYPE_TIME and TYPE_TEXT_LIST as these are the closest types to the standard LANSAs data types A (Alphanumeric), P (Packed) and S (Signed). The data will be converted to the required field type and from EBCDIC to LMBCS for TYPE_TEXT, TYPE_DATE and TYPE_TEXT_LIST fields.

Either an alphanumeric field value or numeric field value should be specified to create the new field. For a TYPE_TIME field, the alphanumeric value may be specified as '*CURRENT', in which case the current date and time will be set for the field otherwise the date/time value must be supplied in the correct format e.g. mm/dd/yy hh:mm:ss. For a TYPE_TEXT_LIST field, the value will be added to the existing field if the text list field already exists.

Note that the document/data note is not updated in the database until you use the DOM_UPDATE_DOCUMENT Built-In Function.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Field name to be added to the document	1	65		

3	N	Req	Field type in note: 1 = TYPE_NUMBER 2 = TYPE_TEXT 3 = TYPE_TIME 4 = TYPE_TEXT_LIST	1	7	0	0
4	A	Opt	Alphanumeric field required for field type TYPE_TEXT, TYPE_TIME & TYPE_TEXT_LIST	1	256		
5	N	Opt	Numeric field required for field type TYPE_NUMBER	1	15	0	9

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Field successfully added. ER = Field not added.	2	2		

Example

Refer to this Domino Built-In Function Example:

[Example 1: Creating a New Document in a Database](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFFieldSetNumber, NSFFieldSetText, ConvertTextToTIMEDATE, NSFFieldSetTime and NSFFieldAppendTextList. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.57 DOM_ADD_ITEM

⇒ **Note:** Built-In Function Rules.

Adds an item to an open data note using the item name. The item type must also be specified.

The item types supported are the simple data types TYPE_TEXT, TYPE_NUMBER, TYPE_TIME and TYPE_TEXT_LIST as these are the closest types to the standard LANSAs data types A (Alphanumeric), P (Packed) and S (Signed). The data will be converted to the required item type and from EBCDIC to LMBCS for TYPE_TEXT, TYPE_DATE and TYPE_TEXT_LIST items.

Either an alphanumeric field value or numeric field value should be specified to create the new item. For a TYPE_TIME item, the alphanumeric value may be specified as '*CURRENT', in which case the current date and time will be set for the item otherwise the date/time value must be supplied in the correct format e.g. mm/dd/yy hh:mm:ss. For a TYPE_TEXT_LIST item, the value will be added to the existing item if the text list item already exists.

Note that the document/data note is not updated in the database until you use the DOM_UPDATE_DOCUMENT Built-In Function.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Item name to be added to the document	1	65		

3	N	Req	Item type in note: 1 = TYPE_NUMBER 2 = TYPE_TEXT 3 = TYPE_TIME 4 = TYPE_TEXT_LIST	1	7	0	0
4	A	Opt	Alphanumeric field required for item type TYPE_TEXT, TYPE_TIME & TYPE_TEXT_LIST Refer to Argument Item 4 .	1	Unlimited		
5	N	Opt	Numeric field required for item type TYPE_NUMBER	1	15	0	9

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Item successfully added. ER = Item not added.	2	2		

Example

Refer to this Domino Built-In Function Example:

[Example 1: Creating a New Document in a Database](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFItemSetNumber, NSFItemSetText, ConvertTextToTIMEDATE, NSFItemSetTime and NSFItemAppendTextList. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

Argument Item 4

This Built-In Function also converts the value of the fourth argument (whenever it is applicable) into LMBCS equivalent using Lotus API OTranslate, which can process up to 65535 bytes only. Therefore, the length of the added Item must not exceed 65535 bytes.

9.58 DOM_CLOSE_DATABASE

⇒ **Note:** Built-In Function Rules.

Closes a previously opened Domino/Notes Database on a local or remote Domino server. If the database resides on a remote server, the session to the server is also closed.

You must use the DOM_CLOSE_DATABASE Built-In Function before exiting your application once the database has been previously opened.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Database handle	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Database successfully closed. ER = Database not closed.	2	2		

Examples

Refer to these Domino Built-In Function Examples:

[Example 1: Creating a New Document in a Database](#)

[Example 2: Selecting documents from a Database using a view](#)

[Example 3: Executing an Agent in a Database](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API NSFDbClose. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.59 DOM_CLOSE_DOCUMENT

⇒ **Note:** Built-In Function Rules.

Closes an open document when no longer required to release Notes allocated memory for the document/data note.

Note that this does not write the contents of the document to the database, you must use the DOM_UPDATE_DOCUMENT Built-In Function to update the document to disk.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Document/Data Note successfully closed. ER = Document/Data Note not closed.	2	2		

Examples

Refer to these Domino Built-In Function Examples in the *Visual LANSA Developer Guide*:

[Example 1: Creating a New Document in a Database](#)

[Example 2: Selecting documents from a Database using a view](#)

[Example 3: Executing an Agent in a Database](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API NSFNoteClose. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.60 DOM_CLOSE_FILE

⇒ **Note:** Built-In Function Rules.

Closes a previously opened Domino/Notes File on a local or remote Domino server. If the file resides on a remote server, the session to the server is also closed.

You must use the DOM_CLOSE_FILE Built-In Function before exiting your application once the file has been previously opened.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	File handle	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = File successfully closed. ER = File not closed.	2	2		

Examples

Refer to these Domino Built-In Function Examples:

[Example 1: Creating a New Document in a Database](#)

[Example 2: Selecting documents from a Database using a view](#)

[Example 3: Executing an Agent in a Database](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API NSFDbClose. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.61 DOM_CREATE_DOCUMENT

⇒ **Note:** Built-In Function Rules.

Creates a new document/data note in memory within an opened database.

Note that the document/data note is empty after creation and is not yet stored in the database. You should use the DOM_ADD_ITEM Built-In Function to add items/fields to the note and the DOM_UPDATE_DOCUMENT Built-In Function to update the document/data note in the database.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Database handle	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Document successfully created. ER = Document not created.	2	2		
2	A	Req	Document/Data Note Handle	4	4		

Example

Refer to these Domino Built-In Function Examples:

[Example 1: Creating a New Document in a Database](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API NSFNoteCreate. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.62 DOM_DELETE_DOCUMENT

⇒ **Note:** Built-In Function Rules.

Deletes a document/data note from the database using the given Note ID.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Database handle	4	4		
2	N	Req	Note ID	1	15	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Document/Data Note successfully deleted. ER = Document/Data Note not deleted.	2	2		

Example

Refer to these Domino Built-In Function Examples:

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API NSFNoteDelete. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.63 DOM_DELETE_FIELD

⇒ **Note:** Built-In Function Rules.

Deletes a field from an open document/data note using the field name. The named field is deleted from the in-memory copy of the document until the DOM_UPDATE_DOCUMENT Built-In Function is used to update the document/note in the database.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Field name in the document whose value is to be deleted.	1	65		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Field value successfully deleted. ER = Field not deleted.	2	2		

		NF = Field value not found.				
--	--	-----------------------------	--	--	--	--

Example

Refer to these Domino Built-In Function Examples:

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFFieldIsPresent and NSFFieldDelete. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.64 DOM_DELETE_ITEM

⇒ **Note:** Built-In Function Rules.

Deletes an item from an open document/data note using the item name. The named item is deleted from the in-memory copy of the document until the DOM_UPDATE_DOCUMENT Built-In Function is used to update the document/note in the database.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Item name in the document whose value is to be deleted.	1	65		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Item value successfully deleted. ER = Item not deleted.	2	2		

		NF = Item value not found.				
--	--	----------------------------	--	--	--	--

Example

Refer to these Domino Built-In Function Examples:

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFItemIsPresent and NSFItemDelete. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.65 DOM_END_SEARCH_DOCS

⇒ **Note:** Built-In Function Rules.

Must be called when the processing of all documents, as the result of the DOM_SEARCH_DOCUMENTS Built-In Function, is complete. This will release all memory that was allocated to process the DOM_SEARCH_DOCUMENTS Built-In Function request.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Note ID Table handle	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Memory successfully released. ER = Error releasing memory.	2	2		

Examples

Refer to these Domino Built-In Function Examples:

[Example 2: Selecting documents from a Database using a view](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API `IDDestroyTable`. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.66 DOM_EXECUTE_AGENT

⇒ **Note:** Built-In Function Rules.

Executes an Agent . The Note ID of the specified agent will be obtained and the Agent will be then be opened and an Agent runtime context will be created before running the Agent.

Agents are design notes that perform custom operations on documents in a database. Agents usually contain formulas that select which documents to process, and calculate values to store in the documents. You may execute any agent that does not depend on Notes user interface functionality.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Database handle	4	4		
2	A	Req	Agent name	1	128		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Agent ran successfully. ER = Agent did not run	2	2		

		successfully.				
--	--	---------------	--	--	--	--

Example

Refer to these Domino Built-In Function Examples:

[Example 3: Executing an Agent in a Database](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NIFFindDesignNote, NIFFindPrivateDesignNote, AgentOpen, AgentCreateRunContext, AgentRedirectStdout, AgentRun, AgentDestroyRunContext and AgentClose. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.67 DOM_GET_FIELD

⇒ **Note:** Built-In Function Rules.

Gets a field from an open data note using the field name. Either an alphanumeric field value or numeric field value will be returned for the field according to the field type.

The field types supported are the simple data types TYPE_TEXT, TYPE_NUMBER, TYPE_TIME and TYPE_TEXT_LIST as these are the closest types to the standard LANSAs data types A (Alphanumeric), P (Packed) and S (Signed). The data will be converted from the required field type and from LMBCS to EBCDIC for TYPE_TEXT, TYPE_TIME and TYPE_TEXT_LIST fields.

For a TYPE_TEXT_LIST field, an entry position must be specified with a value of 0 being for the first entry in the test list field. The return code is set to "NF" when the entry position requested is greater than the number of entries in the text list field.

If a field type of TYPE_TEXT is specified as an argument but the actual field type in the document is TYPE_TEXT_LIST, the first entry in the text list field will be returned by default with no error.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Field name in the document whose value is to be returned.	1	65		

3	N	Req	Field type in note: 1 = TYPE_NUMBER 2 = TYPE_TEXT 3 = TYPE_TIME 4 = TYPE_TEXT_LIST	1	7	0	0
4	N	Opt	Entry position for a TYPE_TEXT_LIST field.	1	15	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Field value successfully retrieved. ER = Field not retrieved. NF = Field value not found.	2	2		
2	A	Opt	Alphanumeric field required for field type TYPE_TEXT, TYPE_TIME & TYPE_TEXT_LIST	1	256		
3	N	Opt	Numeric field required for field type TYPE_NUMBER	1	15	0	9

Examples

Refer to these Domino Built-In Function Examples:

[Example 2: Selecting documents from a Database using a view](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFFieldIsPresent,

NSFFieldInfo, NSFFieldGetNumber, NSFFieldGetText, ConvertTIMEDATEToText, NSFFieldGetTime, NSFFieldGetTextListEntries and NSFFieldGetTextListEntry. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.68 DOM_GET_ITEM

⇒ **Note:** Built-In Function Rules.

Gets an item from an open data note using the item name. Either an alphanumeric field value or numeric field value will be returned for the item according to the item type.

The item types supported are the simple data types TYPE_TEXT, TYPE_NUMBER, TYPE_TIME and TYPE_TEXT_LIST as these are the closest types to the standard LANSAs data types A (Alphanumeric), P (Packed) and S (Signed). The data will be converted from the required item type and from LMBCS to EBCDIC for TYPE_TEXT, TYPE_TIME and TYPE_TEXT_LIST items.

For a TYPE_TEXT_LIST item, an entry position must be specified with a value of 0 being for the first entry in the test list item. The return code is set to "NF" when the entry position requested is greater than the number of entries in the text list item.

If an item type of TYPE_TEXT is specified as an argument but the actual item type in the document is TYPE_TEXT_LIST, the first entry in the text list item will be returned by default with no error.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Item name in the document whose value is to be returned.	1	65		

3	N	Req	Item type in note: 1 = TYPE_NUMBER 2 = TYPE_TEXT 3 = TYPE_TIME 4 = TYPE_TEXT_LIST	1	7	0	0
4	N	Opt	Entry position for a TYPE_TEXT_LIST item.	1	15	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Item value successfully retrieved. ER = Item not retrieved. NF = Item value not found.	2	2		
2	A	Opt	Alphanumeric field required for item type TYPE_TEXT, TYPE_TIME & TYPE_TEXT_LIST	1	256		
3	N	Opt	Numeric field required for item type TYPE_NUMBER	1	15	0	9

Example

Refer to these Domino Built-In Function Examples:

[Example 2: Selecting documents from a Database using a view](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFItemIsPresent,

NSFItemInfo, NSFItemGetNumber, NSFItemGetText, ConvertTIMEDATEToText, NSFItemGetTime, NSFItemGetTextListEntries and NSFItemGetTextListEntry. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.69 DOM_GET_NXT_DOCUMENT

⇒ **Note:** Built-In Function Rules.

Gets the first or next Note ID from the ID Table created by a previous DOM_SEARCH_DOCUMENTS Built-In Function call.

To return the Note ID of the first document in the ID Table set the Previous Note ID argument to 0 otherwise it should be set to the value returned from a previous call to this Built-In Function.

The returned Note ID should then be processed as required by the DOM_OPEN_DOCUMENT, DOM_GET_ITEM, DOM_UPDATE_ITEM, DOM_UPDATE_DOCUMENT,... Built-In Functions.

When the processing of all documents is complete the DOM_END_SEARCH_DOCS Built-In Function must be used.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Note ID Table handle	4	4		
2	N	Req	Previous Note ID: Set to 0 to return the first Note ID in the ID Table otherwise set to the previously returned Note ID.	1	15	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Next Note ID returned successfully. EF = No more Note Ids in the ID Table. ER = Error returning next Note ID.	2	2		
2	N	Req	Note ID returned	1	15	0	0

Examples

Refer to these Domino Built-In Function examples:

[Example 2: Selecting documents from a Database using a view](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API IDScan. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.70 DOM_OPEN_DATABASE

⇒ **Note:** Built-In Function Rules.

Opens a Domino/Notes Database on a local or remote Domino server.

You must use the DOM_CLOSE_DATABASE Built-In Function before exiting your application once the database has been opened.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Notes database file name. Note: This includes the Notes data directory. The file extension may be omitted if it is ".NSF" e.g. "/NOTES/DATA/ACME.NSF"	1	255		
2*	A	Opt	Network port name	1	32		
3*	A	Opt	Domino server name	1	255		

Note: Arguments 2* and 3* required if accessing a Remote Server Directory

To open the database on a local Domino server only requires the database name.

To open the database on a remote Domino server may require a full path name including the server name (TCP/IP host name usually registered in a DNS), the

organisation name, the country code and Notes database name eg.
"SYDNOTES/ACME/AU/NOTES/DATA/ACME.NSF".

The server name may be sufficient if this name exists in the TCP/IP Host Table on the local IBM i server.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Database successfully opened. ER = Database not opened.	2	2		
2	A	Req	Database handle	4	4		

Examples

Refer to these Domino Built-In Function Examples:

[Example 1: Creating a New Document in a Database](#)

[Example 2: Selecting documents from a Database using a view](#)

[Example 3: Executing an Agent in a Database](#)

[Example 4: Updating Documents selected from a Browselist.](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs OTranslate, NotesInit, OSPATHNetConstruct and NSFDbOpen. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.71 DOM_OPEN_DOCUMENT

⇒ **Note:** Built-In Function Rules.

Opens the specified document within a database using the given Note ID.
You must use either the DOM_UPDATE_DOCUMENT or DOM_CLOSE_DOCUMENT Built-In Function when you have finished with the document to release Notes allocated memory.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Database handle	4	4		
2	N	Req	Note ID	1	15	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Document successfully opened. ER = Document not opened.	2	2		
2	A	Req	Document/Data Note Handle	4	4		

Example

Refer to this Domino Built-In Function Examples:

[Example 2: Selecting documents from a Database using a view.](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API NSFNoteOpen. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.72 DOM_OPEN_FILE

⇒ **Note:** Built-In Function Rules.

Opens a Domino/Notes File on a local or remote Domino server.

You must use the DOM_CLOSE_FILE Built-In Function before exiting your application once the file has been opened.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Notes file file name. Note: This includes the Notes data directory. The file extension may be omitted if it is ".NSF" e.g. "/NOTES/DATA/ACME.NSF"	1	255		
2*	A	Opt	Network port name	1	32		
3*	A	Opt	Domino server name	1	255		

***Note:** Arguments 2* and 3* required if accessing a Remote Server Directory

To open the file on a local Domino server only requires the file name.

To open the file on a remote Domino server may require a full path name including the server name (TCP/IP host name usually registered in a DNS), the

organisation name, the country code and Notes file name eg.
"SYDNOTES/ACME/AU/NOTES/DATA/ACME.NSF".

The server name may be sufficient if this name exists in the TCP/IP Host Table on the local IBM i server.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = File successfully opened. ER = File not opened.	2	2		
2	A	Req	File handle	4	4		

Examples

Refer to these Domino Built-In Function Examples:

[Example 1: Creating a New Document in a Database](#)

[Example 2: Selecting documents from a Database using a view](#)

[Example 3: Executing an Agent in a Database](#)

[Example 4: Updating Documents selected from a Browselist.](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs OTranslate, NotesInit, OPathNetConstruct and NSFDbOpen. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.73 DOM_SEARCH_DOCUMENTS

⇒ **Note:** Built-In Function Rules.

Searches a database for documents/data notes matching selection criteria or using a previously created view. If no selection criteria or view is specified then all documents/data notes in the database will match.

The result of this Built-In Function is a Note ID table which can be used to read documents sequentially using the DOM_GET_NXT_DOCUMENT Built-In Function. When selection criteria is specified, the selection formula is compiled and a Note ID table built for all documents matching the criteria. When a view is specified, a collection will be built for the view and all Note Ids for the collection will be added to the Note ID table.

Note that if selection criteria is specified, it must conform to the same syntax as view selection formulas which consist of Notes @functions, field names and logical operators. Selection criteria must be quoted correctly. The syntax of the selection criteria will not be validated.

When the processing of all documents is complete the DOM_END_SEARCH_DOCS Built-In Function must be used.

Some general guidelines for using a view versus using selection criteria are:

- Use selection criteria to select documents from a database when the selection criteria is not known until run time, if the performance of the program is not important or the program will be run only a few times.
- Use a view to select documents from a database when you want to process the documents in a certain order or the performance of the program is important. You will usually find the document selection using a view to be faster than the equivalent selection using selection criteria.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Database handle	4	4		
2	A	Req	Search Type: V = View C = Selection Criteria N = None i.e. all documents will be selected.	1	1		
3	A	Opt	View Name - required if Search Type = V	1	128		
4	A	Opt	Selection Criteria - required if Search Type is C - required if Search Type is V and search is done via the Find By Name method. The value entered is a case insensitive match of the primary sort key.	1	255		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Search completed successfully. ER = Search did not complete successfully.	2	2		
2	A	Req	Note ID Table Handle	4	4		
3	N	Opt	No. Note Ids in the Note ID Table i.e. the number of documents selected.	1	15	0	0

Examples

Refer to these Domino Built-In Function Examples:

[Example 2: Selecting documents from a Database using a view](#)

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs IDCreateTable, NSFFormulaCompile, NSFSearch, IDEntries, IDInsert, NIFFindView, NIFOpenCollection, IDCreateTable, NIFReadEntries and NIFCloseCollection. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.74 DOM_UPDATE_DOCUMENT

⇒ **Note:** Built-In Function Rules.

Updates a document/data note in the database. This writes the in-memory version of the note to the database. Notes allocated memory is also released for the document/data note.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Document/Data Note successfully updated. ER = Document/Data Note not updated.	2	2		

Examples

Refer to these Domino Built-In Function Examples:

[Example 1: Creating a New Document in a Database](#)

[Example 4: Updating Documents selected from a Browselist.](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes API NSFNoteUpdate. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.75 DOM_UPDATE_FIELD

⇒ **Note:** Built-In Function Rules.

Updates an existing field to an open data note using the field name. The field type must also be specified. If the field does not exist it will be added to the document.

The field types supported are the simple data types TYPE_TEXT, TYPE_NUMBER, TYPE_TIME and TYPE_TEXT_LIST as these are the closest types to the standard LANSAs data types A (Alphanumeric), P (Packed) and S (Signed). The data will be converted to the required field type and from EBCDIC to LMBCS for TYPE_TEXT, TYPE_DATE and TYPE_TEXT_LIST fields.

Either an alphanumeric field value or numeric field value should be specified to update the specified field. For a TYPE_TIME field, the alphanumeric value may be specified as '*CURRENT', in which case the current date and time will be set for the field otherwise the date/time value must be supplied in the correct format e.g. mm/dd/yy hh:mm:ss. For a TYPE_TEXT_LIST field, the value will be added to the existing field if the text list field already exists.

Note that the document/data note is not updated in the database until you use the DOM_UPDATE_DOCUMENT Built-In Function.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Field name to be updated in the document	1	65		

3	N	Req	Field type in note: 1 = TYPE_NUMBER 2 = TYPE_TEXT 3 = TYPE_TIME 4 = TYPE_TEXT_LIST	1	7	0	0
4	A	Opt	Alphanumeric field required for field type TYPE_TEXT, TYPE_TIME & TYPE_TEXT_LIST	1	256		
5	N	Opt	Numeric field required for field type TYPE_NUMBER	1	15	0	9

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Field successfully updated. ER = Field not updated.	2	2		

Example

Refer to this Domino Built-In Function Example:

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFFieldSetNumber, NSFFieldSetText, ConvertTextToTIMEDATE, NSFFieldSetTime and NSFFieldAppendTextList. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.76 DOM_UPDATE_ITEM

⇒ **Note:** Built-In Function Rules.

Updates an existing item to an open data note using the item name. The item type must also be specified. If the item does not exist it will be added to the document.

The item types supported are the simple data types TYPE_TEXT, TYPE_NUMBER, TYPE_TIME and TYPE_TEXT_LIST as these are the closest types to the standard LANSAs data types A (Alphanumeric), P (Packed) and S (Signed). The data will be converted to the required item type and from EBCDIC to LMBCS for TYPE_TEXT, TYPE_DATE and TYPE_TEXT_LIST items.

Either an alphanumeric field value or numeric field value should be specified to update the specified item. For a TYPE_TIME item, the alphanumeric value may be specified as '*CURRENT', in which case the current date and time will be set for the item otherwise the date/time value must be supplied in the correct format e.g. mm/dd/yy hh:mm:ss. For a TYPE_TEXT_LIST item, the value will be added to the existing item if the text list item already exists.

Note that the document/data note is not updated in the database until you use the DOM_UPDATE_DOCUMENT Built-In Function.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Document/Data Note handle	4	4		
2	A	Req	Item name to be updated in the document	1	65		

3	N	Req	Item type in note: 1 = TYPE_NUMBER 2 = TYPE_TEXT 3 = TYPE_TIME 4 = TYPE_TEXT_LIST	1	7	0	0
4	A	Opt	Alphanumeric field required for item type TYPE_TEXT, TYPE_TIME & TYPE_TEXT_LIST	1	256		
5	N	Opt	Numeric field required for item type TYPE_NUMBER	1	15	0	9

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Item successfully updated. ER = Item not updated.	2	2		

Example

Refer to this Domino Built-In Function Example:

[Example 4: Updating Documents selected from a Browselist](#)

Technical Notes

This Built-In Function uses the standard Lotus Notes APIs NSFItemSetNumber, NSFItemSetText, ConvertTextToTIMEDATE, NSFItemSetTime and NSFItemAppendTextList. All security and integrity issues related to the use of this Built-In Function are according to normal Lotus Notes API use for the current platform.

9.77 DROP_DD_VALUES

⇒ **Note:** Built-In Function Rules.

Drops a set of dropdown values to free up space for other sets of dropdown values.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Dropdown name Must begin with DD	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code. Returned values possible are: OK: Values dropped successful ER: Error occurred	2	2		

Example

To drop the dropdown values for an asset code field to make room for a status field.

```
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
USE    BUILTIN(ADD_DD_VALUES) WITH_ARGS(DDST '/'
      'Raised/Open/Closed/Invoiced/History/Back Ordered')
      TO_GET(#RETCOD)
IF    COND('#RETCOD *NE OK')

USE    BUILTIN(DROP_DD_VALUES) WITH_ARGS(DDAC)
      TO_GET(#RETCOD)
USE    BUILTIN(ADD_DD_VALUES) WITH_ARGS(DDST '/'
      'Raised/Open/Closed/Invoiced/History/Back Ordered')
      TO_GET(#RETCOD)
ENDIF

USE    BUILTIN(ADD_DD_VALUES) WITH_ARGS(DDST '' 'Cancelled')
      TO_GET(#RETCOD)
IF    COND('#RETCOD *NE OK')
* << error processing >>
ENDIF
```

9.78 DROP_EXTRA_USER_KEYS

⇒ **Note:** Built-In Function Rules.

Disables all "extra" user defined function keys that have been previously enabled by one or more uses of the ALLOW_EXTRA_USER_KEY Built-In Function.

For use with

LANSA for i YES

Visual LANSA for Windows YES

Visual LANSA for Linux YES

Arguments

No arguments.

Return Values

No return values.

9.79 ENCRYPT

⇒ **Note:** Built-In Function Rules.

Encrypt a text string.

A companion Built-in Function, [9.35 DECRYPT](#), is used to decode the encrypted text string.

Warning: From this version (LANSA V11 SP4) onward, a blank key won't be used to encrypt, a generated key will be used instead if the key argument is passed with all blanks. In versions prior to V11 SP4, encrypt will have a key of all spaces.

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	u	Req	Text to be encrypted	8	Unlimited		
2	N	Req	Length of text to be encrypted The value for this length argument must be a multiple of 8. The value provided for this argument must not be greater than the length of Argument 1 (text to be encrypted).	1	11	0	0
3	u	Opt	Key to be used for encryption If a key is not provided for the encryption, a key will be generated and returned. The key used for encryption must be	16	32		

			<p>saved and provided to the DECRYPT Built-In Function.</p> <p>The current encryption cipher uses 16 bytes/128 bits key. The last 16 bytes are reserved for future use.</p> <p>If a Unicode field type is used it is converted to UTF-8 and truncated to 32 bytes. If the key is not provided, and if return value 3 is a Unicode field type, then a Unicode key is generated. This key is converted to UTF-8 and truncated to 32 bytes. Alternatively, a key may be automatically generated using a Unicode field for the key, but setting it to an empty string.</p>				
4	A	Opt	<p>Encrypted text stored in HEX.</p> <p>YES= return encrypted text in HEX format.</p> <p>NO= Return encrypted text in binary format.</p> <p>Default is NO.</p>	2	3		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Returned encrypted text	8	Unlimited		
2	A	Opt	<p>Return code</p> <p>OK = action completed</p> <p>ER = error occurred</p>	2	2		
3	u	Opt	Key used for encryption when no key argument provided	16	32		

Technical Notes

- Cipher block encryption algorithms always encrypt and decrypt blocks of 8 characters. Therefore the actual value you encrypt must have a length **that is a multiple of 8**.
- The encrypted data returned by the encryption algorithm is binary data and can contain any value. As such, using it or passing it through environments where it may be subject to code page conversion (eg: database managers, communications links, etc) or where special characters like CR, LF or binary zero may cause issues (eg: HTML or XML documents, string processing, stream file processing, etc) may prove to be problematic. As such it is recommended that encrypted data is physically stored in hexadecimal format.
- Unicode fields are converted to UTF-8 before encryption. This allows a Unicode field to be encrypted on one platform and decrypted on another. It also means that the encryption length (argument 2) can be up to 3 times the length of the text to be encrypted (argument 1). And then if the result is stored in hex it's further doubled in size. So if you have a 500 character NVarchar then the encrypted length should be 1500 and the size of the returned encrypted text should be 3000. If you do not have the lengths in this ratio then data may be lost.

Examples

Following are two RDML subroutines that demonstrate a generic encryption technique for any information up to 16 characters in length.

For example EXECUTE ENCRYPT (#KEY #PROD_NUM #PROD_ENC) might encrypt #PROD_NUM (char 10) to produce encrypted #PROD_ENC (char 32).

Note that even though the subroutine parameter #PROD_NUM is a char 10 field, the resulting encryption field #PROD_ENC is char 32.

This is because the initial binary encryption requires an input that is a multiple of 8 (ie: char 10 #PROD_NUM is padded with blanks to be 16 bytes long by the subroutine execution) and it produces a hexadecimal representation of the encrypted binary data, which is therefore 32 bytes long.

All 32 bytes of #PROD_ENC need to be stored for successful decryption to occur later.

To decrypt these values you would code EXECUTE DECRYPT (#KEY

#PROD_ENC #PROD_NUM).

Here the 32 byte hexadecimal value is first converted to binary, decrypted and then returned as a char 16.

The subroutine finally returns the decrypted value right truncated into #PROD_NUM as a char 10.

ENCRYPT a value with a supplied key and return a 32 byte encrypted value in hex

```
***** =====
***** Sample routine to Encrypt a passed in value (up to 16
***** bytes in length) with a supplied key and return a
***** 32 byte encrypted value in hex (suitable for storing in
***** database, etc)
***** =====
SUBROUTINE NAME(ENCRYPT) PARMS((#KEY16 *RECEIVED) (#VAL
DEFINE   FIELD(#KEY16) TYPE(*CHAR) LENGTH(16) DESC('Encryptic
DEFINE   FIELD(#VAL16) TYPE(*CHAR) LENGTH(16) DESC('Value to b
DEFINE   FIELD(#HEX32) TYPE(*CHAR) LENGTH(32) DESC('Encrypted
DEFINE   FIELD(#LEN) TYPE(*DEC) LENGTH(5) DECIMALS(0)
CHANGE   #LEN 16
***** Use ENCRYPT BIF to encrypt #VAL16 of length #LEN using
***** #KEY16 to return encrypted value in #HEX32
***** The encrypted value is converted into HEX resulting in
***** a 32 byte value.
USE      BUILTIN(ENCRYPT) WITH_ARGS(#VAL16 #LEN #KEY16 YES)
ENDROUTINE
***** =====
```

DECRYPT a Hex value using the supplied key and return the unencrypted value

```
***** =====
***** Sample routine to Decrypt a passed in Hex value
***** using the supplied key and return the unencrypted
***** value.
***** =====
SUBROUTINE DECRYPT ((#DKEY16 *Received)(#DHEX32 *Received)
(#DVAL16 *Returned))
***** Key to be used for the decryption. This must be the
***** same key that was used for the encryption.
```

```

DEFINE #DKEY16 *char 16
DEFINE FIELD(#DHEX32) TYPE(*CHAR) LENGTH(32) DESC('Encrypt
DEFINE FIELD(#DVAL16) TYPE(*CHAR) LENGTH(16) DESC('Decrypt
DEFINE FIELD(#DLEN) TYPE(*DEC) LENGTH(5) DECIMALS(0)
CHANGE FIELD(#DLEN) TO(16)
*****
*****
***** Use DECRYPT BIF to decrypt character #HEX32 of length
***** #DLEN using #DKEY16 to return decrypted value,#DVAL16
*****
USE BUILTIN(DECRYPT) WITH_ARGS(#HEX32 #DLEN #DKEY16 Y
ENDROUTINE

```

ENCRYPT a password and then DECRYPT

```

DEFINE FIELD(#PASSWORD) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#TEXT) TYPE(*CHAR) LENGTH(16)
DEFINE FIELD(#LENGTH) TYPE(*DEC) LENGTH(11) DECIMALS(0)
DEFINE FIELD(#KEY) TYPE(*CHAR) LENGTH(16)
DEFINE FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#ENCRYPTED) TYPE(*CHAR) LENGTH(16)
DEFINE FIELD(#DECRYPTED) TYPE(*CHAR) LENGTH(16)
*****
***** Encrypt password with key
CHANGE #TEXT #PASSWORD
CHANGE #LENGTH 16
CHANGE #KEY 'AXG12345lj0gtUMX'
USE BUILTIN(ENCRYPT) WITH_ARGS(#TEXT #LENGTH #KEY) TC
*****
***** Decrypt password with same key as provided for encryption
CHANGE #LENGTH 16
USE BUILTIN(DECRYPT) WITH_ARGS(#ENCRYPTED #LENGTH #K
*****

```

9.80 END_FILE_EDIT

⇒ **Note:** Built-In Function Rules.

Ends an "edit session" on the definition of a nominated LANSa file definition previously started by the START_FILE_EDIT Built-In Function.

The edit session may have been used to define a new file or alter an existing one.

The file definition is released by this Built-In Function so that it can be accessed by other users.

A number of checks that relate to prior actions via the LOGICAL_KEY and ACCESS_RTE_KEY Built-In Functions are performed via this function. These may result in the abandonment of the edit session, and an "ER" return code being returned.

Additionally, warning messages may be issued by this Built-In Function. In this case the return code will still be returned as "OK".

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSa for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSa for Windows	YES	The access route key field validation performed by this Built-In Function is not as rigorous as that performed by LANSa for i.
Visual LANSa for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Commit edited details	1	1		

		flag				
		Y = commit details				
		N = drop edited details				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = edit session ended ER = fatal error detected In case of "ER" return code error message(s) are issued automatically and the edit session ended without commitment	2	2		

9.81 END_FUNCTION_EDIT

⇒ **Note:** Built-In Function Rules.

Ends an active edit session on a LANSAs function definition.

An edit session is commenced by using the Built-In Function START_FUNCTION_EDIT.

A function edit session should be terminated by using the END_FUNCTION_EDIT Built-In Function to ensure all locks/etc are released/shutdown in an orderly manner.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No argument values.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = edit session ended	2	2		

		ER = fatal error detected				
--	--	------------------------------	--	--	--	--

9.82 END_PROCESS_EDIT

⇒ **Note:** Built-In Function Rules.

Ends an active edit session on a LANSAs process definition

An edit session is commenced using the Built-In Function START_PROCESS_EDIT.

A process edit session should be terminated using the END_PROCESS_EDIT Built-In Function to ensure all locks/etc are released/shutdown in an orderly manner.

Any process edit session that receives a fatal error will have an END_PROCESS_EDIT command automatically issued.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No argument values.

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = edit session	2	2		

		ended	ER = fatal error detected				
--	--	-------	---------------------------	--	--	--	--

9.83 END_RTV_SPLF_LIST

⇒ **Note:** Built-In Function Rules.

Used in conjunction with START_RTV_SPLF_LIST and GET_SPLF_LIST_ENTRY.

The START_RTV_SPLF_LIST must be used first to provide the selection criteria for the retrieval of spool files. Once the selection criteria are established, the GET_SPLF_LIST_ENTRY can be used to retrieve the details of the spool files.

The END_RTV_SPLF_LIST must be used after the list of spool files have been retrieved. This will close the list and release the storage allocated to that list.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return code. OK = The list was closed successfully.	2	2		

Example

Refer to [9.130 GET_SPLF_LIST_ENTRY](#) for an example.

9.84 EXCHANGE_ALPHA_VAR

⇒ **Note:** Built-In Function Rules.

Places an alphanumeric variable / value onto the exchange list

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Field name	1	10		
2	A	Req	Value	1	Unlimited		

Return Values

This Built-In Function does not return values.

Example

This function can be used to exchange a variable to another function, with a different name. That is, exchange the value of field #CODEDES, but give the field a different name, e.g. #DESC.

```
***** Retrieve code description
```

```
*****
```

```
FETCH  FIELDS(#CODEDES) FROM_FILE(CDMST)
```

```
*****
```

```
***** Exchange CODEDES as DESC
```

```
*****
```

```
USE    BUILTIN(EXCHANGE_ALPHA_VAR) WITH_ARGS('DESC' #CC
```

```
CALL    PROCESS(PPPPPPPP) FUNCTION(FFFFFFFF)
```

This Built-In Function can also be used in place of the EXCHANGE command, i.e. to simply exchange a value.

```
EXCHANGE FIELDS(#VALUE)  
CALL    PROCESS(PPPPPPPP) FUNCTION(FFFFFFFF)
```

is functionally identical to the following:

```
USE     BUILTIN(EXCHANGE_ALPHA_VAR) WITH_ARGS('VALUE' #V  
CALL    PROCESS(PPPPPPPP) FUNCTION(FFFFFFFF)
```

9.85 EXCHANGE_NUMERIC_VAR

⇒ **Note:** Built-In Function Rules.

Places a numeric variable/value onto the exchange list.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Platform Notes

Note the difference between the maximum length of the value:
30.9 in IBM i and 63.63 in Visual LANSA

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Field name	1	10		
2	N	Req	Value	1	Unlimited		Unlimited

Return Values

No values are returned by this Built-In Function.

Examples

This function can be used to exchange a variable to another function, with a different name. That is, exchange the value of field #PRDTOTAL but give the field a different name, e.g. #TOTAL.

```
***** Calculate product total
```

```
*****
```

```
SELECT  FIELDS(#COST) FROM_FILE(PRDMST)
```

```
KEEP_TOTAL OF_FIELD(#COST) IN_FIELD(#PRDTOTAL)
ENDSELECT
*****
***** Exchange PRDTOTAL as TOTAL
*****
USE    BUILTIN(EXCHANGE_NUMERIC_VAR)
      WITH_ARGS('TOTAL' #PRDTOTAL)
CALL   PROCESS(PPPPPPP) FUNCTION(FFFFFFF)
```

This Built-In Function can also be used in place of the EXCHANGE command, i.e. to simply exchange a value.

```
EXCHANGE FIELDS(#PRICE)
CALL     PROCESS(PPPPPPP) FUNCTION(FFFFFFF)
```

is functionally identical to the following:

```
USE    BUILTIN(EXCHANGE_NUMERIC_VAR)
      WITH_ARGS('PRICE' #PRICE)
CALL   PROCESS(PPPPPPP) FUNCTION(FFFFFFF)
```

9.86 EXCHANGE_VARIABLE

⇒ **Note:** Built-In Function Rules.

Places a variable/value onto the exchange list.

For use with

LANSA for i	YES. Only available with RDMLX.
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Field name	1	10		
2	X	Req	Value	1	Unlimited		Unlimited

Return Values

No values are returned by this Built-In Function.

9.87 EXECUTE_TEMPLATE

⇒ **Note:** Built-In Function Rules.

Executes an application template to generate RDML function code into a working list

Generated RDML code is appended to the END of the working list, so the list may need to be cleared before (via the CLR_LIST command) before invoking the application template.

Alternatively, multiple templates may be executed serially to progressively build up the resulting RDML function code.

This Built-In Function can only be used against a function that has been previously placed into an edit session using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a commercial application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of Application Template to be	1	10		

		executed. This template must be previously defined using the LANSAs Application Template facilities.				
--	--	--	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		
2	L	Req	Working list Name. The working list must be formatted as described in the 9.110 GET_FUNCTION_RDML Built-In Function and must not contain more than 32767 entries.				

9.88 EXPONENTIAL

⇒ **Note:** Built-In Function Rules.

Performs exponentiation by raising a base number to an exponent.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Base digit portion Decimal portion and digits beyond 13th position ignored.	1	15	0	9
2	N	Req	Base decimal portion Digit portion and decimals beyond 5th position ignored.	1	15	0	9
3	N	Req	Exponent digit portion Decimal portion and digits beyond 13th position ignored.	1	15	0	9
4	N	Req	Exponent decimal portion Digit portion and decimals beyond 5th position ignored.	1	15	0	9
5	A	Opt	Rounding of result required: Values: Y or N Default: N Note: rounding is at the 5th decimal position.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Result digit portion Accurate to 13 digits.	1	15	0	9
2	N	Req	Result decimal portion Accurate to 5 decimals.	1	15	0	9
3	A	Opt	Return code Values: Y = good return N = error occurred	1	1		

Examples

Calculate 2.345 raised to the power of 5.67 accurate to five decimal positions:

```
USE    BUILTIN(EXPONENTIAL) WITH_ARGS(2 0.345 5 0.67)
      TO_GET(#DIGITS #DECIMALS)
CHANGE FIELD(#RESULT) TO('#DIGITS + #DECIMALS')
```

Read a packed decimal 15,5 base and a packed decimal 5,2 exponent from a workstation and display the result to the user as a packed decimal 15,5:

```
DEFINE FIELD(#BASE) TYPE(*DEC) LENGTH(15) DECIMALS(5) LAI
DEFINE FIELD(#EXPN) TYPE(*DEC) LENGTH(5) DECIMALS(2) LAB
DEFINE FIELD(#RDGT) TYPE(*DEC) LENGTH(10) DECIMALS(0)
DEFINE FIELD(#RDEC) TYPE(*DEC) LENGTH(5) DECIMALS(5)
DEFINE FIELD(#RSLT) TYPE(*DEC) LENGTH(15) DECIMALS(5) LAE
REQUEST FIELDS(#BASE #EXPN)
USE    BUILTIN(EXPONENTIAL) WITH_ARGS(#BASE #BASE
#EXPN #EXPN) TO_GET(#RDGT #RDEC)
CHANGE FIELD(#RSLT) TO('#RDGT + #RDEC')
DISPLAY FIELDS(#BASE #EXPN #RSLT)
```

9.89 EXPORT_OBJECTS

⇒ **Note:** Built-In Function Rules.

Creates LANSa Import formatted files for all LANSa objects specified in an input list.

The input list contains the object types and names to be exported. (Optionally, the whole partition can be exported.) For each object in the list, information in the related internal tables will be unloaded in LANSa Import format. As each list entry is processed a completion message is written to the file export.log that will be automatically created / replaced in the temporary directory. The message will indicate if the definition for the object was successfully exported or if it failed. If any one definition fails to export successfully the return code will be set to ER.

Portability Considerations This BIF cannot be used for exporting development source to/from a Linux platform.

For use with

LANSa for i	NO
Visual LANSa for Windows	YES
Visual LANSa for Linux	YES

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working List of Objects to have internal data exported. Formatted: Start End Description 1 - 2 Object Type where : AA = Technology Service	32	32		

DF = Field/Component/WAM				
FD = File				
PD= Process				
PF = Function				
SV = System Variable				
MT = Multilingual Variable				
AT = Template				
RM = Message/Message File				
BI = Built-In Function				
WL = Weblet				
WC=Web Component (HTML)				
XC=Web Component(XML)				
3 - 32 Details of the object				
For AA				
1 - 10 Entity ID				
11 - 20 Technology Service ID				
For DF				
1 - 10 Field/Component Name				
For FD				
1 - 10 File Name				
11 - 20 File Library				
21 - 30 Library Substitution Item				
For PD				
1 - 10 Process Name				
For PF				
1 - 10 Process Name				
11 - 17 Function Name				
For SV/Weblet				
1 - 20 System Variable Name				
For BI				
1 - 20 BIF Name				
For AT				
1 - 10 Template Name				
For MT				
1 - 20 Multilingual Variable Name				
For RM				
1 - 4 Language Code				
5 - 14 Message File				
15 - 21 Message Id				
For WL				
1 - 20 Weblet Name				
For WC and XC				
1 - 20 Web Component Name				
21-25 Secondary Extension Name				

			26-26 Input or Output Visual Note: If Message Id is left blank all messages for the message file language will be extracted.				
2	A	Req	Export directory NB. This directory MUST exist.	256	256		
3	A	Opt	Export Whole Partition Y - Ignore list passed and export all objects in the partition N - Process the entries in the list of objects to be exported. Default = N	1	1		
4	A	Opt	Append to Existing Files Y - Append export data to any existing export files in the export directory N - Replace any export files in the export directory Default = Y	1	1		
5	A	Opt	Export System Definition (LX_F46/LX_F96) Y - System Definitions exported N - System Definitions not exported Default - Y	1	1		
6	A	Opt	Export to Development System Y - include internal data for a development system (eg. RDML source) Cannot be set to Y if exporting to a Linux platform. N - do not include internal data for a development system Default - N	1	1		
7	A	Opt	Reset Build Status Y - Reset the exported object's build status to	1	1		

			<i>Build and Compile Required.</i> N - Export the object's build status as it is. Default - N				
8	List	Opt	Library Directive File Substitutions This list will contain the details for the PARTITION and USERLIB information to be put in LXXLDF: 1 - 4 Prompt Language 5 - 14 Override Level 15 - 24 Override Item 25 - 44 Override Value 45 - 45 Prompt Override 46 - 195 Prompt Text	195	195		
9	A	Opt	Silent Mode. Y – Perform the export object definitions without showing the Log Window. Default = N	1	1		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code OK = The export completed without error ER = An Error occurred during the export.	2	2		

Technical Notes

The Export to Development System and Export System Definition flags control

the amount of information exported. If Export to Development System is set to N (the default), this will only export definitions that are required for execution. If Export to Development System is set to Y, this will export the full definitions (e.g. RDML for functions, access routes for files) for the various objects. Assuming all other required objects are exported (e.g. fields on the file) or already available in the target environment, this will allow development of the objects on the target system.

Some definitions will only be exported if the Export to Development System flag is set to Y. For example, BIFs, and templates, which are not required on a non-development system.

To export standard definitions to a development system, set both Export to Development System and Export System Definition to Y. This will export definitions for all BIFs, Templates, RDML commands, etc.

9.90 FETCH_IN_SPACE

⇒ **Note:** Built-In Function Rules.

Fetches the first cell row that matches the key values supplied and returns the cell values into the specified fields.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name	1	256		
2-20	w	O	Fields that specify the key values to be used to locate the first cell row required.	1	Unlimited	0	Unlimited

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "OK" = A cell row was found and the cell values have been returned. "NR" = No cell row could be found with a key matching the key	2	2		

			values supplied. "ER" = Select attempt failed. Messages issued will indicate more about the cause of the failure.				
2-20	w	O	Fields to receive the values of the cells in the selected cell row.	1	Unlimited	0	Unlimited

Technical Notes

The return fields must be specified in the same order as the cells in the space were defined. Cells are matched by the order of their specification in return values 2 -> 20. The names of the fields used have no bearing whatsoever on the cell mapping logic.

You can specify less key values than are defined in the space. The first matching cell row will be returned.

If you specify more key values than are defined as key cells for the space then the additional values will be ignored and have no effect on the outcome of the search.

If you specify less return field values than there are cells in the space then the non-specified cells are not mapped back into the fields.

If you specify more return field values than there are cells in the space then the additional field values are ignored and are not changed by the search operation.

If a key value longer than 256 bytes is specified, a fatal error will occur.

9.91 FILE_FIELD

⇒ **Note:** Built-In Function Rules.

FILE_FIELD specifies or re-specifies a field that is part of the record format of the file definition being edited.

An edit session must be commenced by using the START_FILE_EDIT Built-In Function prior to using this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Warning: The FILE_FIELD Built-In Function cannot be used for a file of type "OTHER".

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of real field. Must be defined in the LANSA data dictionary.	1	10		
2	N	Opt	Optional sequencing number. Used to order fields within the file record format. If not specified fields are sequenced in the same order as they are presented.	1	5	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = field added to file ER = fatal error detected In case of "ER" return code error message(s) are issued automatically and the edit session ended without commitment	2	2		

9.92 FILE_FIELD_VIRTUAL

⇒ **Note:** Built-In Function Rules.

Specifies or re-specifies a virtual field that is part of the definition of the file being edited.

An edit session must be commenced by using the START_FILE_EDIT Built-In Function prior to using this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of the virtual field. Must be defined in the LANSA data dictionary.	1	10		
2	N	Opt	Optional sequencing number. Used to order fields within the file record format. If not specified fields are sequenced in the same order as they are presented.	1	5	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
----	------	-------------	-------------	------------	------------	------------	------------

1	A	Req	Return code OK = field added to file ER = fatal error detected In case of "ER" return code error message(s) are issued automatically and the edit session ended without commitment	2	2		
---	---	-----	---	---	---	--	--

9.93 FILLSTRING

⇒ **Note:** Built-In Function Rules.

Fills a field with as many occurrences of a specified string as will fit in a given field.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be repeated	1	Unlimited		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Returned string	1	Unlimited		

Note: This function will put as many occurrences of a string as will fit. If the target field length is not a multiple of the length of the string to be repeated, the last occurrence will be truncated.

Examples

Fill a field with the character '*' (asterisk).

```
DEFINE FIELD(#OUTTEXT) TYPE(*CHAR) LENGTH(10)
```

```
USE    BUILTIN(FILLSTRING) WITH_ARGS("'*") TO_GET(#OUTEXT)
DISPLAY FIELDS(#OUTEXT)
```

Resulting display would look something like this:

```
FUN01    Fillstring BIF

Out text . . . *****

CF1=Help
```

Fill a string with a requested value.

```
DEFINE  FIELD(#INTEXT) TYPE(*CHAR) LENGTH(4)
DEFINE  FIELD(#OUTEXT) TYPE(*CHAR) LENGTH(18)
*****
REQUEST FIELDS(#INTEXT)
USE    BUILTIN(FILLSTRING) WITH_ARGS(#INTEXT) TO_GET(#OUTEXT)
DISPLAY FIELDS(#OUTEXT)
```

Resulting displays would look something like this:

```
FUN01    Fillstring BIF

In text . . . FRED

CF1=Help
```

then,

```
FUN01    Fillstring BIF

Out text . . . FREDFREDFREDFREDFR

CF1=Help
```


9.94 FINDDATE

⇒ **Note:** Built-In Function Rules.

Finds the date that is 'n' days after/before a given date.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Date to be counted from	6	8	0	0
2	N	Req	No. of days(+/-) after/before the given date. Note: For non-IBM i systems, this value should be less than 214783648 and greater than -214783649, otherwise a fatal execution error will occur.	1	15	0	0
3	A	Opt	Date format of given date Default: A	1	1		
4	A	Opt	Date format of returned date Default: A	1	1		

Valid Date Formats

These date formats are valid formats for given and returned dates: A, B, D, F, H,

J, L, V and 1.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Returned date	6	8	0	0
2	A	Opt	Returned okay code (Y/N)	1	1		

Note: All dates must have a four character year so that accurate comparisons and calculations can be performed. Where a two character year (e.g. DDMMYY, YYMMDD, MMY) is supplied the century value is retrieved from the system definition data area. The year supplied is compared to a year in the data area, if the supplied year is less than or equal to the comparison year then the less than century is used. If the supplied year is greater than the comparison year then the greater than century is used.

Example

Find the date field #NXTDAT in date format YYMMDD (D) that is #NUMD days after date field #DMY in date format DDMMYY (B):

```
USE      BUILTIN(FINDDATE) WITH_ARGS(#DMY #NUMD B D)
        TO_GET(#NXTDAT)
```

9.95 FINDDATE_ALPHA

⇒ **Note:** Built-In Function Rules.

Finds the date that is 'n' days after or before a given date.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Date to be counted from	1	10		
2	N	Req	No. of days(+/-) after/ before the given date. Note: For non-IBM i systems, this value should be less than 214783648 and greater than -214783649, otherwise a fatal execution error will occur.	1	15	0	0
3	A	Opt	Date format of given date Default: A	1	1		
4	A	Opt	Date format of returned date Default: A	1	1		

Valid Date Formats

These date formats are valid formats for given and returned dates: A, B, C, D,

E, F, G, H, I, J, K, L, M, V and 1.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Returned date	1	10		
2	A	Opt	Returned okay code (Y/N)	1	1		

Note: All dates must have a four character year so that accurate comparisons and calculations can be performed. Where a two character year (e.g. DDMMYY, YYMMDD, MMY) is supplied the century value is retrieved from the system definition data area. The year supplied is compared to a year in the data area, if the supplied year is less than or equal to the comparison year then the less than century is used. If the supplied year is greater than the comparison year then the greater than century is used.

Example

Find the date field #NXTDAT in date format YYMMDD (D) that is #NUMD days after date field #DMY in date format DDMMYY (B):

```
USE      BUILTIN(FINDDATE_ALPHA) WITH_ARGS(#DMY #NUMD B D)
        TO_GET(#NXTDAT)
```

9.96 FORMAT_STRING

⇒ **Note:** Built-In Function Rules.

This Built-in Function returns a character string which is built from an input Format Pattern. The Format Pattern can consist of text plus field values. Editing options may be applied to the field values.

Special Note: All characters preceded by a colon (:) will be treated as a field name and ends with either one of the following characters () space and colon.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Format Pattern</p> <p>Refer to Technical Notes for pattern syntax and options.</p> <p>Fields specified as :field</p> <p>Optional formatting options may be appended to fields. The option/s immediately follow the field name and are enclosed in brackets ().</p> <p>(editcode,x)</p> <p>apply edit code to field value where x is a valid LANSA editcode. Refer to Standard Field Edit Codes for a list of valid edit codes.</p>	1	Unlimited		

			<p>(substr,n1,n2) apply substring to field value where n1=start position, n2=length.</p> <p>(triml) remove leading blanks from field value.</p> <p>(trim) remove trailing blanks from field value.</p> <p>(trimall) remove leading and trailing blanks from field value.</p> <p>(upper) convert field value to uppercase.</p> <p>(lower) convert field value to lowercase.</p>				
2	A	Opt	<p>DBCS enable</p> <p>Default: disable</p> <p>YES = to enable</p> <p>Note:</p> <p>For substring, the number count is for each character, not byte.</p>	3	3		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Formatted String	1	unlimited		
2	A	Opt	<p>Return code</p> <p>OK = action completed</p> <p>ER = Error occurred</p>	2	2		

Technical Notes

- Field names are preceded by a colon (:).
Example Format Pattern =
Employee :givenname :surname with number :EMPNO was not found.

Return Formatted String =
Employee DARREN BROWN with number A0001 was not found.

when field givenname contains the value "DARREN", field surname contains the value "BROWN" and field empno contains the value "A0001"

- Fields used in the Format Pattern must be used elsewhere in the RDML function.
Fields may be defined in the repository or internally within the function.
- To include a colon (:) in the returned Formatted String, use two colons in the input Format pattern.
Example Format Pattern=
Employee no:::empno

Returned Formatted String=
Employee no:A0001

- To include a character straight after the field, use brackets().
Example Format Pattern=
Employee no:::empno()NoSpace

Returned Formatted String=
Employee no:A0001NoSpace

- Formatting option/s may be applied to a field value and specified in the Format Pattern.
Multiple options may be applied to a field value.
Formatting options must immediately follow the field name and be contained within brackets ().

Example Format Pattern=
Employee :givenname(substr,1,1 UPPER). :SURNAME(trim upper)

Returned Formatted String=
Employee D. BROWN

- When multiple formatting options are specified for a field value, the options are applied in the following order
 1. edit code
 2. substring
 3. trim trailing (trim)
 4. trim leading (triml)
 5. trim all
 6. lower
 7. upper
- The Edit code formatting option are applied only to numeric fields. If an edit code formatting option is specified for a character field, it will be ignored.
- Edit codes which suppress leading zeroes will remove any resulting leading blanks.

DBCS considerations

Text sections of the Format Pattern may contain DBCS or mixed characters. However the :field specification must be entered in single byte mode. Also any text section must be a complete string with the correct pairing of shift out/ shift in characters.

The substring format option is not DBCS sensitive by default. To enable DBCS, set the second optional argument to 'YES'. Note that the number for the start position and length are in character count, not in byte and the shift in and shift out bytes are not counted.

Example

This example retrieves information from a file and formats different lines into a standard browse list. A variety of formatting options are used to format field values.

```
FUNCTION  OPTIONS(*DIRECT)
DEFINE   FIELD(#STRING) TYPE(*CHAR) LENGTH(75) COLHDG('Deti
DEFINE   FIELD(#PATERN) TYPE(*CHAR) LENGTH(256) INPUT_ATR(
DEF_LIST NAME(#BRWLST) FIELDS((#STRING))
*****
SELECT   FIELDS((#EMPNO) (#GIVENAME) (#SURNAME) (#STARTDT
CHANGE   FIELD(#PATERN) TO('"EMPLOYEE:: :empno :GIVENAME(st
```

```

EXECUTE SUBROUTINE(ADDTOBRW)
CHANGE FIELD(#PATERN) TO("      Start :startdte(editcode ,Y) Salary
EXECUTE SUBROUTINE(ADDTOBRW)
CHANGE FIELD(#PATERN) TO("      Address:: :ADDRESS1(trim) :AL
EXECUTE SUBROUTINE(ADDTOBRW)
CHANGE FIELD(#PATERN) TO("      :aADDRESS3(trima
ll) :postcode(editcode,4)")
EXECUTE SUBROUTINE(ADDTOBRW)
ENDSELECT
*****

DISPLAY BROWSELIST(#BRWLST)
RETURN
***** -----
SUBROUTINE NAME(ADDTOBRW)
USE BUILTIN(FORMAT_STRING) WITH_ARGS(#PATERN) TO_GET
ADD_ENTRY #BRWLST
ENDROUTINE
.
```

9.97 GET_AUTHORITIES

⇒ **Note:** Built-In Function Rules.

Retrieves a list of authorities to LANSAs objects and returns it to the calling RDML function in a variable length working list.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML applications.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object name.	1	10		
2	A	Req	Object extension.	1	10		
3	A	Req	Object type. Valid types are: AT - Application template DF - Field FD - File PD - Process PF - Function P# - Partition	2	2		

			SV - System variable MT - Multilingual variable				
4	A	Opt	User name.	1	10		

Dependencies

If any of Object type, Object name or Object extension are specified, then all three must be specified according to the following table.

Object Type	Object Name	Object Extension
AT	template name	*blank
DF	field name	*blank
FD	file name	*blank, *LIBL, library name
PD	process name	*blank
PF	process name	function name
P#	partition name	*blank
SV	positions 1-10 of system variable name	positions 11-20 of system variable name
MT	positions 1-10 of multilingual variable name	positions 11-20 of multilingual variable name

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = list returned partially or completely filled. No more authorities that match the	2	2		

			<p>arguments exist</p> <p>OV = list returned completely filled, but more authorities that match the arguments than could fit in the list exist.</p> <p>NR = No authorities that match the arguments exist. Last entry in the list is returned as null.</p> <p>ER = Error in the arguments passed. Last entry in the list is returned as null.</p>				
2	L	Req	<p>Working list of authorities.</p> <p>If Object type, name and extension are specified but not User, then as many authorities of users to the object as fit in the list will be returned.</p> <p>If User is specified but not Object type, name and extension, then as many authorities of the user to different objects as fit in the list will be returned.</p> <p>If Object type, name, extension and User are specified, and the user is specifically authorized to the object then the authority of the user to the object will be returned in the list.</p> <p>If the user is not specifically authorized to the object no authorities will be returned.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 70 bytes.</p> <p>List cannot be more than: 32767 entries in Windows 9999 entries on IBM i.</p> <p>From - To Description 1 - 10 Object name 11 - 20 Object extension 21 - 22 Object type (see above for object type explanation) 23 - 32 User name 33 - 52 Access rights</p>	70	70		

		This is a string of 2 character codes representing the different access rights that the user has to the object.			
--	--	---	--	--	--

The individual access rights are:

UD - Use Definition

MD - Manage Definition

DD - Existence of Definition

DS - Data - Display

AD - Data - Add

CH - Data - Change

DL - Data - Delete

If the entire string is blank then the user has had their access rights to the object revoked.

53 - 70 <<future expansion>>

9.98 GET_BIF_LIST

⇒ **Note:** Built-In Function Rules.

Searches for the BIF name and returns a working list containing user defined BIF details.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No.	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Search Built-In Function Name	1	20		

Return Values

No.	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing BIF details. The list must be in the following format and not contain more than 32,767 entries: From - To Description 1 - 4 BIF Number 5 - 24 BIF Name	64	64		

			25 - 64 BIF Description				
2	A	Req	The last Built-In Functions in the returned list.	1	20		
3	A	Req	<p>Return Code</p> <p>OK = list returned partially or completely filled with BIF details. No more Built-In Functions exist beyond those returned in the list.</p> <p>OV = list returned completely filled, but more Built-In Functions than could fit in the list still exist.</p> <p>NR = list was returned empty. Last Built-In Function in list returned as blanks.</p>	2	2		

9.99 GET_CHAR_AREA

⇒ **Note:** Built-In Function Rules.

Gets a character string from a character data area.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Data area name	1	10		
2	A	Opt	Library name Default: *LIBL When data area is *LDA or *GDA this argument must be *LIBL	1	10		
3	A	Opt	Lock data area Y - lock data area N - do not lock data area Default: N When data area is *LDA or *GDA this argument is ignored.	1	1		
4	N	Opt	Start pos. to retrieve from Default: position 1	1	5	0	0
		Req	If *LDA or *GDA is the data area				
5	N	Opt	Length to retrieve Default: full length	1	4	0	0
		Req	If *LDA or *GDA is the data area				

Note: Start position and length, if specified, must BOTH be provided as argument values.

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Returned value	1	2000		

Examples

Retrieve a company name from a data area named COMPID. Use the job's current library list to locate the data area:

```
USE BUILTIN(GET_CHAR_AREA) WITH_ARGS(COMPID) TO_GET(#CC
```

Retrieve bytes 101 - 110 of data area called USERINFO in library QTEMP and place the result into a field called #OUTQ:

```
USE BUILTIN(GET_CHAR_AREA)
```

```
WITH_ARGS(USERINFO QTEMP N 101 10) TO_GET(#OUTQ)
```

Warning: When processing *LDA or *GDA, a start position and a length **must** be specified. If these arguments are not specified the program will terminate abnormally. * Retrieve the first 30 bytes of information from the Local Data Area (*LDA). **Note:** The *LDA data area will not be locked even if it is specified in the functions arguments.

```
DEFINE FIELD(#RETVAL) TYPE(*CHAR) LENGTH(30)
```

```
USE BUILTIN(GET_CHAR_AREA) WITH_ARGS(" *LDA " " *LIBL " N 1
```

Retrieve some information passed by one of my group jobs into the Group Data Area (*GDA). The information is in positions 20 to 50 of the *GDA:

```
DEFINE FIELD(#RETVAL) TYPE(*CHAR) LENGTH(30)
```

```
USE BUILTIN(GET_CHAR_AREA) WITH_ARGS('*GDA' '*LIBL' N 2
```

9.100 GET_COMPONENT_LIST

⇒ **Note:** Built-In Function Rules.

Returns a list of Components. All simple fields will be ignored by this BIF. This will be done by processing one of the LANSAs internal tables that list the components in one of the three selection methods. It will order the components by component name if the search type is by component, or by Group and component name if the search type is by Group, or by Framework and component if the search type is by Framework.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Use

If all the search parameters are blank a search by component will be assumed.

By Component

To search by component the user will be required to provide a value for the Search Component parameter and leave Search Framework/Group blank.

To continue loading the list with subsequent data after receiving a return code of OV set the Search Component parameter to the value of the Last Component return value with Search Group/Framework set to blank. The BIF will use this as the pointer from which it will read the next set of information requested.

By Group/Framework

To search by group/framework the user will be required to provide a value for the Search Group/Framework parameter and leave Search Component.

To continue loading the list with subsequent data after receiving a return code of OV set the Search Group/Framework and Search Component parameters to the

value of Last Group/Framework and Last Component return values. The BIF will use this as the pointer from which it will read the next set of information requested.

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Search Type The type of search required. Valid values: CMP = by Component FRW = by Framework GRP = by Group	3	3		
2	A	Opt	Search Component Positioning Component value.	1	10		
3	A	Opt	Search Framework/Group Positioning Framework/Group	1	20		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing Component details. The list must be in the following format and not contain more than 32,767 entries: From - To Description 1 - 10 Component Name 11 - 50 Component Description 51 - 70 Framework	90	90		

			71 - 90 Group				
2	A	Req	The last Component in the returned list.	1	10		
3	A	Req	The last Group/Framework in the returned list.	1	20		
4	A	Req	Return Code OK = list returned partially or completely filled with component details. No more Components exist beyond those returned in the list. OV = list returned completely filled, but more Components than could fit in the list still exist. NR = list was returned empty. Last Component/Framework/Group in list returned as blanks.	2	2		

9.101 GET_COMPOSITION

⇒ **Note:** Built-In Function Rules

Returns the list of objects that comprise a LANSAs Object for all currently enabled build environments. An enabled environment is indicated by `envenab=YES` in `x_bldenv.dat`. Note that the Microsoft Windows environment (`envtype=W32`) is returned as `W95` as only one or the other can be enabled at one time and they share the same directory structure and filenames.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	YES

Processing

This BIF will return the list of Objects that comprise a specific object name and type. It will get the information from `X_BLDENV.DAT` and return the fully calculated value of the file name and path. It also returns a single entry with the object's generated C name for ALL platforms.

Arguments

No.	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	LANSAs Object	1	20		
2	A	Req	Object Type: DF = Field/Component FD = File PD = Process	2	2		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	<p>Composition list for an object.</p> <p>The list must be in the following format and not contain more than 32,767 entries:</p> <p>From-To Description</p> <p>1 - 5 Platform (Linux, etc)</p> <p>6 - 7 Object Type</p> <p> CN = Generated C Name</p> <p> EO = Executable</p> <p> SO = Source</p> <p>8 - 27 Object Name (Physical name of the file)</p>	27	27		
2	A	Req	<p>Return Code</p> <p>OK = The list was returned without error</p> <p>ER = An Error occurred during the getting of the list.</p>	2	2		

9.102 GET_ENVIRONMENTS

⇒ **Note:** Built-In Function Rules.

Returns a list of the Environment Names and a Flag to indicate if the environment is enabled for build purposes from X_BLDENV.DAT. Environment name will be returned from the envname value and the enabled flag will be determined from the evenab value.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

None

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing Environment details. The list must be in the following format and not contain more than 32,767 entries: From - To Description 1 - 5 Environment Type 6 - 75 Environment Name 76 - 76 Environment Enabled	76	76		

9.103 GET_FIELD

⇒ **Note:** Built-In Function Rules.

Retrieves attributes of a field stored in the LANSAs Repository and returns them to the calling RDML function.

Returned values are exactly as presented in [Detailed Display of a Field Definition](#) in the *LANSAs for i User Guide*.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of field to be retrieved from Repository	1	10		
2	A	Opt	Name of process or *COMP if component *COMP is not available in LANSAs for i in non-RDMLX partitions.	1	10		
3	A	Opt	Name of function or component. Compulsory if Arg 2 is provided. If this argument is provided, the field in Arg 1 will be looked for first in the working fields for this function or component, and if not found as a working field, then the data dictionary will be searched. Note that the working field definitions for the function or component are as of the most recent compile.	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = field details returned ER = field not accessible In case of "ER" return code error message(s) are issued automatically.	2	2		
2	A	Opt	Field type A = alpha S = signed decimal numeric P = packed decimal numeric B = Binary E = Date F = Float H = Char I = Integer M = Time Z = DateTime 1 = String 2 = VarBinary 3 = CLOB 4 = BLOB 8=Nchar 9=NVarChar	1	1		
3	N	Opt	Length of field or total number of digits in field. Length will be zero for types BLOB and CLOB.	3	15	0	0
4	N	Opt	Number of decimal positions Not applicable to all types.	1	15	0	0
5	A	Opt	Reference field name	1	10		
6	A	Opt	Field description	1	40		

7	A	Opt	Field label	1	15		
8	A	Opt	List of 3 * A(20) headings Bytes 1-20 are column head 1. Bytes 21-40 are column head 2. Bytes 41-60 are column head 3.	1	60		
9	A	Opt	List of 10 * A(4) output attributes	1	40		
10	A	Opt	List of 10 * A(4) input attributes	1	40		
11	A	Opt	Edit code or edit word. If first char is a quote (') then value is an edit word, otherwise it is an edit code. Not applicable to type A field.	1	20		
12	A	Opt	Default value of field	1	20		
13	A	Opt	Optional alias name of field	1	30		
14	A	Opt	System field flag YES = a system field NO = not a system field	3	3		
15	A	Opt	Keyboard shift	1	1		
16	A	Opt	Component (Y/N) Note: Fields which have visualization return N.	1	1		
17	A	Opt	Definition source W: Working field D: Data dictionary. Will always be 'D' if process & function arguments not supplied.	1	1		
18	A	Opt	Prompting process name Return Blank for working field	1	10		
19	A	Opt	Prompting function name Return Blank for working field	1	7		
20	A	Opt	Is field RDMLX?	1	1		

		Y=Field is RDMLX N=Field is RDML.				
--	--	--------------------------------------	--	--	--	--

9.104 GET_FIELD_INFO

⇒ **Note:** Built-In Function Rules.

Retrieves a list of field related information from the LANSAs internal database and returns it to the calling RDML function in variable length working lists.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Field name	1	10		
2	A	Req	Level at which information is requested D = Dictionary level F = File level	1	1		
3	A	Req	Type of field related information to retrieve. Valid types are : FIELDCHECK - Validation rules MLATTR- Multilingual attributes	1	10		

			REFFLD - Fields referring to the requested field				
4	A	Opt	Physical file name. Required if file level information is requested.	1	10		
5	A	Opt	Physical file library. Required if file level information is requested.	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = list returned partially or completely filled. No more of this type of information exists for this field. OV = list returned completely filled, but more of this type of information than could fit in the list exists. NR = list was returned empty. Last entry in the list is returned as null. ER = Field not found. Last entry in the list is returned as null.	2	2		
2	L	Req	<i>Header working list</i> to contain field related information. The calling RDML function must: - provide a working list with an aggregate entry length of exactly 100 bytes. - contain no more than: 32,767 entries if Windows 9999 entries if IBM i . For type of field related information to retrieve see Format for header working list type	100	100		

			<p>FIELDCHECK: Format for header working list type</p> <p>MLATTR: Format for header working list type</p> <p>REFFLD: Format for header working list type</p>				
3	L	Req	<p><i>Detail working list</i> to contain field related information.</p> <p>The calling RDML function must provide a working list with:</p> <ul style="list-style-type: none"> - an aggregate entry length of exactly 100 bytes - contain no more than: 32,767 entries if Windows 9999 entries if IBM i . <p>Refer to:</p> <p>Format for detail working list type</p> <p>FIELDCHECK: Format for detail working list type</p> <p>MLATTR: Format for detail working list type</p> <p>REFFLD: for details.</p>	100	100		

Format for header working list type FIELDCHECK:

Bytes	Description
1-5	Number of the first entry in the detail list for this entry in character format. A value of 00000 denotes that there are no entries in the detail list for this entry.
6-10	Number of the last entry in the detail list for this entry in character format
11-12	Type of check. SL = Simple Logic, DC = Date Check, CF = File Check, CL = Complex Logic, RV = Range of Values, LV = List of Values
13-42	Description of check.
43-43	Enable check for ADD. Y = Check performed on ADD, U = Check performed on ADD USE, N = Check not performed on ADD.

44-44	Enable check for CHANGE. Y = Check performed on CHG, U = Check performed on CHG USE, N = Check not performed on CHG.
45-45	Enable check for DELETE. Y = Enable check, N = Do not enable check.
46-46	Action if check is true. N = Perform NEXT check, E = Issue fatal ERROR, A = ACCEPT value and do no more checking.
47-47	Action if check is false. N = Perform NEXT check, E = Issue fatal ERROR, A = ACCEPT value and do no more checking.
48-54	Error Message Number.
55-64	Message File Name.
65-74	Message File Library.
75-84	Name of program to be called to perform Complex Logic check. Blank if not Complex Logic check.
85-94	Name of file used in File check. Blank if not File check.

Format for header working list type MLATTR:

Bytes	Description
1-5	Number of the first entry in the detail list for this entry in character format. A value of 00000 denotes that there are no entries in the detail list for this entry.
6-10	Number of the last entry in the detail list for this entry in character format
11-14	Language code.
15-29	Label.
30-49	Column heading 1.
50-69	Column heading 2.
70-89	Column heading 3.

Format for header working list type REFFLD:

Bytes	Description
1-5	A value of 00000 since the detail list is not used for REFFLD information.
6-10	A value of 00000.
11-20	Field Name
21-60	Description

Format for detail working list type FIELDCHECK:

Bytes	Description
Type of check Simple Logic	
Each detail list entry is formatted as follows:	
1-79	Condition line.
Type of check Date Check	
One detail list entry is formatted as follows:	
1-8	Date format.
9-15	Number of days allowed into the past for specified date in character format.
16-22	Number of days allowed into the future for specified date in character format.
Type of check File Check	
Each detail list entry is formatted as follows:	
1-20	Value used as a key to the file.
Type of check Complex Logic	

Each detail list entry is formatted as follows:

1-20	Value used as an additional parameter.
------	--

Type of check Range of Values

Each detail list entry is formatted as follows:

1-20	Value used as low limit of range.
------	-----------------------------------

21-40	Value used as high limit of range.
-------	------------------------------------

Type of check List of Values.

Each detail list entry is formatted as follows:

1 - 20	Value used as a list element.
--------	-------------------------------

Format for detail working list type MLATTR:

Bytes	Description
Each detail list entry is formatted as follows:	
1-40	Field description.

Format for detail working list type REFFLD:

Bytes	Description
No information is returned in the detail list.	

9.105 GET_FIELD_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of fields and their descriptions from the Repository and returns them to the calling RDML function in a variable length working list.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Positioning field value. The returned list starts with the first field from the dictionary whose name is greater than the value passed in this argument.	1	10		
2	A	Opt	Omit Visual LANSA Components From list Y/N. Default is N	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec

1	L	Req	<p>Working list to contain Field information.</p> <p>The calling function must provide a working list with:</p> <ul style="list-style-type: none"> - an aggregate entry length of exactly 60 bytes - no more than 9999 entries. <p>Each returned list entry is formatted as follows:</p> <p>From-To Description</p> <p>1 - 10 Field Name</p> <p>11 - 50 Field Description</p> <p>51 - 51 RDMLX Field (Y or N)</p> <p>52-60 <<future expansion>></p>	60	60		
2	A	Opt	<p>Last field in returned list. Typically this value is used as the positioning argument on subsequent calls to this Built-In Function.</p>	1	10		
3	A	Opt	<p>Return code.</p> <p>OK = list returned partially or completely filled with field details. No more fields exist beyond those returned in the list.</p> <p>OV = list returned completely filled, but more fields than could fit in the list exist. Typically used to indicate "more" fields in page at a time style list displays.</p> <p>NR = list was returned empty. Last field in the list is returned as blanks.</p>	2	2		

Example

A user wants to customize some field definitions by changing labels, and column headings.

```

FUNCTION OPTIONS(*DIRECT)
GROUP_BY     NAME(#FLDDTL) FIELDS((#FLDNAM *NC) (#FLDDES *
DEF_LIST     NAME(#FLDLST) FIELDS(#FLDNAM #FLDDES #SPARE)
*****
-Request field-

```

```
REQUEST    FIELDS(#STRFLD) TEXT(('Field to start from' 5 5))
*****    -Get list of fields-
USE        BUILTIN(GET_FIELD_LIST) WITH_ARGS(#STRFLD) TO_GET
*****    -Process lists-
SELECTLIST NAMED(#FLDLST)
USE        BUILTIN(GET_FIELD) WITH_ARGS(#FLDNAM) TO_GET(#R
*****    < break column headings into FLDCH1, FLDCH2, FLDCH3 >
*****    -Change field definition-
REQUEST    FIELDS(#FLDDTL)
USE        BUILTIN(PUT_FIELD) WITH_ARGS('NNN' #FLDNAM #FLDT
TO_GET(#STD_CMPAR)
ENDSELECT
```

9.106 GET_FILE_INFO

⇒ **Note:** Built-In Function Rules.

Retrieves a list of file related information from the LANSAs internal database and returns it to the calling RDML function in variable length working lists.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Physical file name	1	10		
2	A	Req	Physical file library (value ignored by CHECKFILE). In Visual LANSAs blanks are also validated for backward compatibility.	1	10		
3	A	Req	Type of file related information to retrieve. Valid types are: CHECKFILE – First library the file exists in.	1	10		

		<p>FIELDS- Fields in the file</p> <p>VIRTUALS – Virtual fields in the file.</p> <p>PHYKEYS- Fields used as keys to the file.</p> <p>LGLVIEWS- Logical views for the file.</p> <p>ACCROUTES- Access routes for the file.</p> <p>MLATTR- Multilingual attributes</p>				
--	--	--	--	--	--	--

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return code</p> <p>OK = lists returned partially or completely filled. No more of this type of information exists for this file.</p> <p>OV = lists returned completely filled, but more of this type of information than could fit in the list exists.</p> <p>NR = details list was returned empty. (FIELDS/VIRTUALS return OK)</p> <p>ER = File not found</p>	2	2		
2	L	Req	<p>Header working list to contain file related information.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 100 bytes.</p> <p>List cannot be more than: 32767 entries in Windows 9999 entries on IBM i.</p> <p>From - To Description</p> <p>1 - 5 Number of the first entry in the detail list for this entry in character format. A value of '00000' denotes that there are</p>	100	100	0	0

no entries in the detail list for this entry.

6 - 10 Number of the last entry in the detail list for this entry in character format

11 - 100 Rest of information

For type CHECKFILE:

One header list entry formatted as follows:

From - To Description

1 - 5 As above

6 - 10 As above

11 - 20 File name

For type FIELDS:

One header list entry formatted as follows:

From - To Description

1 - 5 As above

6 - 10 As above

For type VIRTUALS:

One header list entry formatted as follows:

From - To Description

1 - 5 As above

6 - 10 As above

For type PHYKEYS:

One header list entry is formatted as follows:

From - To Description

1 - 5 As above

6 - 10 As above

For Type LGLVIEWS:

Each header list entry is formatted as follows:

From - To Description

1 - 5 As above

6 - 10 As above

11 - 20 Logical view name

21 - 60 Logical view description

0 0

			<p>For type ACCROUTES :</p> <p>Each header list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 5 As above</p> <p>6 - 10 As above</p> <p>11 - 20 Access route name</p> <p>21 - 60 Access route description</p> <p>61 - 70 File accessed</p> <p>71 - 80 File library accessed.</p> <p>81 - 84 (P7,0) Maximum records expected.</p> <p>For type MLATTR:</p> <p>Each header list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 5 As above</p> <p>6 - 10 As above</p> <p>11 - 20 Logical view name or physical file name</p>			
3	L	Req	<p>Detail working list to contain file related information.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 50 bytes.</p> <p>For type CHECKFILE:</p> <p>Single detail list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 10 First library name that the file exists in</p> <p>For type FIELDS:</p> <p>Each detail list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 10 Field name that is part of the file</p> <p>For type VIRTUALS:</p> <p>Each detail list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 10 Virtual field name that is part of the file</p>	50		

11 - 11 Virtual field populates real field on output
(Yes/No/Unknown)

12 - 12 Real field populates virtual field on input
(Yes/No/Unknown).

For type PHYKEYS:

Each detail list entry is formatted as follows:

From - To Description

1 - 10 Field name that is part of the file key

For type LGLVIEWS:

Each detail list entry is formatted as follows:

From - To Description

1 - 10 Field name that is part of the logical view key

For type ACCROUTES :

Each detail list entry is formatted as follows:

From - To Description

1 - 20 Value that is used as a key in the access route.

For type MLATTR:

Each detail list entry is formatted as follows:

From - To Description

1 - 4 Language code

5 - 44 Logical view or physical file description

9.107 GET_FUNCTION_ATTR

⇒ **Note:** Built-In Function Rules.

Gets an attribute of a function definition that is being edited within an edit session previously started by using the START_FUNCTION_EDIT Built-In Function.

Attributes set or returned by this Built-In Function have the same editing and validation rules as the equivalent online facility provided in a full LANSAs development environment.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of attribute to get Valid attribute names are: DESC- Function description ONMENU - Display on Menu MENUSQ - Menu sequence Number	1	10		

		TOTCMD - Total RDML commands (including comments)				
		EDTSRC - Associated editing source				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		
2	A	Req	Area in which to return the attribute. Allowable values are as follows For attribute DESC: The function description up to 40 characters in length. For attribute ONMENU: Y – Displayed on Menu N – Not displayed on Menu For attribute MENUSQ: Valid number represented as 5 characters. Range 00001 to 99999. For attribute TOTCMD: Valid number represented as 4 characters. Range 0000 to 9999. For attribute EDTSRC: Character 3 editing source as the "source" field on the START_PROCESS_EDIT Built- In Function. Value LAN or blanks indicates last editor was standard online RDML editor	1	256		

9.108 GET_FUNCTION_INFO

⇒ **Note:** Built-In Function Rules.

Retrieves a list of function related information from the LANSAs internal database and returns it to the calling RDML function in a variable length working list.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Process name.	1	10		
2	A	Req	Function name.	1	7		
3	A	Req	Type of function related information to retrieve. Valid type for all LANSAs systems: MLATTR- Multilingual attributes Valid types for LANSAs on IBM i :	1	10		

		<p>FIELDS- Fields used by the function</p> <p>FILES- Files used by the function</p> <p>FUNCPANEL - LANSAs Documentor panel layouts</p> <p>FUNSTEXT- LANSAs Documentor function MSL Diagram</p> <p>FUNTNOTE- LANSAs Documentor function MSL technical notes</p> <p>FUNTABLE- LANSAs Documentor function MSL tables</p> <p>FUNXR3GL- LANSAs Documentor called 3GL programs</p> <p>FUNXRPRO- LANSAs Documentor called processes</p> <p>FUNXRFUN- LANSAs Documentor called functions</p> <p>FUNXRBIF- LANSAs Documentor called Built-In Functions</p> <p>FUNCREP- LANSAs Documentor report layouts functions</p> <p>FUNXRSYV- LANSAs Documentor system variables used</p> <p>FUNXRMST- LANSAs Documentor message text used</p> <p>FUNXRMSI- LANSAs Documentor predefined messages used</p>				
--	--	--	--	--	--	--

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code	2	2		

			<p>OK = list returned partially or completely filled. No more of this type of information exists for this function.</p> <p>OV = list returned completely filled, but more of this type of information than could fit in the list exists.</p> <p>NR = list was returned empty. Last entry in the list is returned as null.</p> <p>ER = Function not found. Last entry in the list is returned as null.</p>				
2	L	Req	<p>Working list to contain process related information.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 132 bytes.</p> <p>List must not be more than: 32767 entries in Windows 9999 entries on IBM i.</p> <p>For type FIELDS: Each returned list entry is formatted as follows</p> <p>From - To Description 1 - 10 Field name. 11 - 132 <<future expansion>></p> <p>For type FILES: Each returned list entry is formatted as follows</p> <p>From - To Description 1 - 10 Physical file name 11 - 20 Physical file library 21 - 30 Logical view name (blank if the physical file used) 31 - 132 <<future expansion>></p> <p>For type FUNXXXXXXXXXX: (ex LANSA Documentor) Each returned list entry is formatted as</p>	132	132		

follows

From - To Description

1 - 132 LANSAs Documentor line.

For type MLATTR:

Each returned list entry is formatted as follows

From - To Description

1 - 4 Language code

5 - 44 Function description

45 - 132 <<future expansion>>

9.109 GET_FUNCTION_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of processes, associated functions and their descriptions from the LANSAs internal database and returns them to the calling RDML function in a variable length working list

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Process name.	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain function information. List must not be more than: 32767 entries in Windows 9999 entries on IBM i. The calling RDML function must provide a working list with an aggregate entry length of	60	60		

			<p>exactly 60 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 7 Function name</p> <p>8 - 47 Function description</p> <p>48 - 48 RDMLX function (Y or N)</p> <p>49-60 <<future expansion>></p>				
2	A	Opt	<p>Return code.</p> <p>OK = list returned partially or completely filled with function details. No more functions exist for this process.</p> <p>OV = list returned completely filled, but more functions than could fit in the list exist. Typically used to indicate "more" functions in page at a time style list displays.</p> <p>NR = list was returned empty. Last function in the list is returned as blanks.</p> <p>ER = Process not found. Last function in the list is returned as blanks.</p>	2	2		

9.110 GET_FUNCTION_RDML

⇒ **Note:** Built-In Function Rules.

Returns the RDML code associated with a function into a working list.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No argument values.

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		
2	L	Req	Working list Name.				

		<p>If the edit session is Include RDML audit stamps N, then the working list must be no greater than 32,767 entries and have an aggregate entry length of 72 bytes where each entry is composed of:</p> <p>From - To Description</p> <p>1 - 4 Command sequence number Format Signed (4, 0).</p> <p>5 - 7 Command Label A(3)</p> <p>8 - 17 Command</p> <p>18 - 72 Command Parameters. Alpha.</p> <p>If the edit session is Include RDML audit stamps Y, then the working list must have an aggregate entry length of 99 bytes, where, in addition to the positions and number of entries described for Include RDML audit stamps N, each entry is composed of:</p> <p>From-To Description</p> <p>73 - 73 Command Changed Flag, always N.</p> <p>74 - 81 Command Changed Date. Signed(8,0) (CCYYMMDD).</p> <p>82 - 91 Command Changed User. Alpha.</p> <p>92 - 99 Command Changed Task. A(8)</p>			
--	--	--	--	--	--

Technical Notes

Commands that have more than 55 bytes of parameters are returned in multiple entries like this example

```

Seq Lab Command Parameters
0001 ***** This is a comment line
0002 SET_MODE TO(*CHANGE)
0003 L32 GROUP_BY NAME(#GROUP) FIELDS(#FIELD001 #FIELD002
0004 #FIELD003 #FIELD004 #FIELD005 #FIELD006)
0005 DISPLAY FIELDS(#GROUP)
0006 MENU
0006 ***** This is a comment line

```

9.111 GET_HELP

⇒ **Note:** Built-In Function Rules.

Gets a list of help text for a specified field, function or process.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object name The name of a field, function or process.	1	10		
2	A	Req	Object extension name If the object type is a function then this value should contain the name of the process that the function is defined in. If the object type is not a function then this value should be blank.	1	10		
3	A	Req	Object type Values: DF - Field PD - Process PF - Function	2	2		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain help text. List must not be more than: 32767 entries in Windows 9999 entries on IBM i. The calling RDML function must provide a working list with an aggregate entry length of exactly 77 bytes. Each returned list entry is formatted as follows: Bytes 1-77: Help Text	1	77		
2	A	Req	Return code OK = list returned partially or completely filled with help text for this object No more help text exists for this object. OV = list returned completely filled, but more help text than could fit in the list exist. Typically used to indicate "more" functions in page at a time style list displays. ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued automatically.	2	2		

Example

A user wants to retrieve the help text of a specific object and display it without the use of the HELP key.

```
***** Define arguments and lists
DEFINE FIELD(#OBJNAM) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#OBJEXT) TYPE(*CHAR) LENGTH(10)
```

```

DEFINE FIELD(#OBJTYP) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#HLPTXT) TYPE(*CHAR) LENGTH(77)
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
DEF_LIST NAME(#WKHLPL) FIELDS((#HLPTXT)) TYPE(*WORKING)
DEF_LIST NAME(#BWHLPL) FIELDS((#HLPTXT))
GROUP_BY NAME(#RQSOBJ) FIELDS((#OBJNAM) (#OBJEXT)
(#OBJTYP))
GROUP_BY NAME(#DSPHLP) FIELDS((#OBJNAM) (#OBJEXT)
(#OBJTYP))
***** Clear working and browse lists
BEGIN_LOOP
***** Request Object Name, Extension and Type
REQUEST FIELDS(#RQSOBJ)
CLR_LIST NAMED(#WKHLPL)
CLR_LIST NAMED(#BWHLPL)
***** Execute built-in-function - GET_HELP
USE BUILTIN(GET_HELP) WITH_ARGS(#OBJNAM #OBJEXT
#OBJTYP)
TO_GET(#WKHLPL #RETCOD)
***** Help text was retrieved successfully
IF COND(#RETCOD *EQ "OK")
***** Move Help text from the working list to the browselist
SELECTLIST NAMED(#WKHLPL)
ADD_ENTRY TO_LIST(#BWHLPL)
ENDSELECT
***** Allow Help text to be reviewed for the specified object
DISPLAY FIELDS((#DSPHLP)) BROWSELIST(#BWHLPL)
***** Working list overflowed, more help text to retrieve
ELSE
IF COND(#RETCOD *EQ "OV")
MESSAGE MSGTXT('List not big enough to fit all help text')
***** GET_HELP failed with errors, report error
ELSE
MESSAGE MSGTXT('GET_HELP failed with errors, try again')
ENDIF
ENDIF
END_LOOP

```

9.112 GET_IENTRY_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of Impact List entries and their descriptions from the data dictionary and returns them to calling RDML function in a variable length working list. The Impact List must have been previously created and entries added using the LANSAs development menu.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSAs for i	YES	Not available for RDMLX.
Visual LANSAs for Windows	NO	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Impact List name from which entries are to be retrieved.	7	7		
1	A	Req	Start positioning value consisting of Entry type and Entry Name. The returned list starts with the first entry from the Impact List which is greater than the value passed in this argument. Entry types are : DF (field), FD (file), PF (function), SV (system variable), MT (multilingual text) Entry names are : Field - field name	1	22		

		File - file name and library Function - process and function names System variable name Multilingual variable name.				
--	--	--	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	<p>Working list to contain Entry information. The calling RDML function must provide a working list with an aggregate entry length of exactly 70 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 2 ENTRY TYPE</p> <p>DF = field FD = file PF = function SV = system variable MT = multilingual variable</p> <p>3 - 22 ENTRY NAME</p> <p>Field</p> <p>3 - 12 Field name 13 - 22 Blank</p> <p>File</p> <p>3 - 12 File name 13 - 22 Library</p> <p>Function</p> <p>3 - 12 Process name 13 - 19 Function name 20 - 22 Blank</p> <p>System variable</p> <p>3 - 22 Variable name</p>				

			Multilingual Variable 3 - 22 Variable name 23 - 62 DESCRIPTION 63 - 70 << FUTURE EXPANSION>>				
2	A	Opt	Last entry in returned list Typically this value is used as the positioning argument on subsequent calls to this Built-In Function.	1	22		
3	A	Opt	Return code. OK = list returned partially or completely filled with Impact List entry details. No more entries exist beyond those returned in the list. OV = list returned completely filled, but more entries than could fit in the list exist. NR = list was returned empty. Last entry in the list is returned as blanks.	2	2		

Example

This function could be used to compile the file entries, which exist in an Impact List. The Impact List would have been created and the file entries added using the LANSAs developer's menu option "Work with Impact Lists".

```

***** #ETYP *CHAR 2
***** #FILE *CHAR 10
***** #LIB *CHAR 10
***** #DESC *CHAR 40
***** #SPARE *CHAR 8
***** #START *CHAR 22
***** #LAST *CHAR 22
DEF_LIST NAME(#ELLST) FIELDS(#ETYP #FILE #LIB #DESC #SPAF
TYPE(*WORKING) ENTRYS(10)
***** -Clear list-
CLR_LIST NAMED(#ELLST)
***** -Request Impact List name-
REQUEST FIELDS(#ILNAME) TEXT(('Impact List to use' 5 5))

```

```

***** -Set the start value to start at the file entries-
CHANGE    FIELD(#START) TO(FD)
***** -Get the entries from the Impact List-
BEGIN_LOOP
USE       BUILTIN(GET_IENTRY_LIST) WITH_ARGS(#ILNAME #STA
        TO_GET(#ELLST #LAST #RETCOD)
***** -If entries found-
IF        COND('#RETCOD *EQ OK) *OR (#RETCOD *EQ OV))
SELECTLIST NAMED(#ELLST)
IF        COND('#TYP = FD')
USE       BUILTIN(MAKE_FILE_OPERATIONL) WITH_ARGS(#FILE #I
        TO_GET(#RTN)
ENDIF
ENDSELECT
***** -If more entries, set start value for repeat -
IF        COND('#RETCOD *EQ OV')
CHANGE    FIELD(#START) TO(#LAST)
ELSE
RETURN
ENDIF
*****
***** -No entries-
ELSE
RETURN
ENDIF
END_LOOP

```

9.113 GET_KEYWORD_STRING

⇒ **Note:** Built-In Function Rules.

Gets the keywords and their values from a string containing one ESF (external source format) statement.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for WINDOWS	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing 'lines' of string from which keywords and their values will be retrieved. There is a limit of 1000 characters in total for this list.	1	256		
2	N	Req	The length of working list entry for the 'lines' of the string from which keywords and their values will be retrieved.	1	3	0	0
3	A	Req	Tag name (command name) of the ESF statement that is being processed.	1	10		
4	L	Req	Working list of keywords to search for. The calling RDML function must provide a working list with an aggregate entry length of exactly 16 bytes. Each list entry is formatted as follows: From - To Description 1 - 15 Keyword	16	16		

		16 - 16 Number of Values: S - Single Value L - List of Values				
--	--	---	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working List for keywords found. The calling RDML function must provide a working list with an aggregate entry length of exactly 25 bytes. Each list entry is formatted as follows: From - To Description 1 - 15 Keyword 16 - 20 First value list entry number for keyword 5,0 Signed 21 - 25 Last value list entry number for keyword 5,0 Signed	25	25		
2	L	Req	Working List for values found The calling RDML function must provide a working list with an aggregate entry length of exactly 131 bytes. Each list entry is formatted as follows: From - To Description 1 - 1 Value Type: A - Alphanumeric N - Numeric 2 - 101 Alpha value. Note: The alpha value is always enclosed in quotes. 102 - 131 Numeric value. Note: The numeric literal is a 30, 9 signed value.	131	131		
3	L	Req	Working list for the leftover part of the searched string after the search keywords and their values have been removed.	1	22		

		Each list entry is formatted as follows:				
		From - To Description				
		1 - 2 Error code . Error codes are listed below.				
		3 - 22 The first 20 character string that was not recognized.				

Error codes

- 01 Tag open delimiter was not a colon
- 02 Tag name does not start with an alpha character
- 03 Tag name too long - must be less/equal to 10 characters
- 04 Keyword does not start with an alpha character
- 05 Keyword Invalid - not found in declared keyword list
- 06 Keyword incomplete - end of string found
- 07 Keyword specified too long - must be less/equal to 15 characters
- 08 No Values specified for keyword
- 09 Value specified is too long
- 10 Multiple values specified for a single value list
- 11 Quoted value does not end in a quote
- 12 Invalid numeric literal value
- 13 More than one decimal format character specified for value
- 14 Digit portion of numeric literal value is longer than 21
- 15 Decimal portion of numeric literal is longer than 9
- 16 Command string longer than allowed maximum of 1000 characters
- 17 Tag close delimiter not specified
- 18 Value incomplete - end of string found
- 19 Expected tag name not found

20 Quoted value must be followed by a blank

21 End of keyword relator not specified

22 Keyword specified more than once

Technical Notes

- The returned keywords list will have an entry for each keyword searched for, in the order specified in the keywords to search for list. A keyword not found will have 0 as its first value list entry number.
- Ensure all keywords specified in the working list of keywords to be searched for and the keywords specified within the ESF statement are in UPPERCASE.
- The alpha value in the returned values list will always be there whether the value is alpha or numeric. The numeric value will only be nonzero if the value is numeric.
- Alphanumeric values that contain lowercase characters and that are not enclosed in quotes will be converted to UPPERCASE.
- Alphanumeric values must not contain embedded quotes.
- The maximum length allowable for an alphanumeric value is 98, all alphanumeric values will be returned in quotes.
- The string in the returned leftover list will consist of the search string where an error has occurred.

Example

A list has been constructed containing an ESF style statement. It has been determined that it is the RECORD statement. The value for FILENAME which is a single value keyword is required.

```
DEFINE FIELD(#KWD) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#KWDTYPE) TYPE(*CHAR) LENGTH(1)
DEFINE FIELD(#LINE) TYPE(*CHAR) LENGTH(70)
DEFINE FIELD(#KWDSTR) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEFINE FIELD(#KWDEND) TYPE(*DEC) LENGTH(5) DECIMALS(0)
DEFINE FIELD(#VALTYPE) TYPE(*CHAR) LENGTH(1)
DEFINE FIELD(#VALALPHA) TYPE(*CHAR) LENGTH(50)
DEFINE FIELD(#VALNUM) TYPE(*DEC) LENGTH(30) DECIMALS(0)
DEFINE FIELD(#FILENAME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#LEFTCOUNT) TYPE(*DEC) LENGTH(5) DECIMALS(0)
```

```

DEF_LIST NAME(#KWDSRCH) FIELDS((#KWD) (#KWDTYPE))
    TYPE(*WORKING)
DEF_LIST NAME(#STRSRCH) FIELDS((#LINE)) TYPE(*WORKING)
DEF_LIST NAME(#KWDFND) FIELDS((#KWD) (#KWDSTR) (#KW DEN
    TYPE(*WORKING)
DEF_LIST NAME(#VALFND) FIELDS((#VALTYPE) (#VALALPHA)
    (#VALNUM)) TYPE(*WORKING)
DEF_LIST NAME(#STRLEFT) FIELDS((#LINE)) TYPE(*WORKING)
    COUNTER(#LEFTCOUNT)
***** Construct list containing ESF:RECORD statement
. . . . .
***** Clear the keyword search list
CLR_LIST NAMED(#KWDSRCH)
***** Put in search keywords
CHANGE FIELD(#KWD) TO(FILENAME)
CHANGE FIELD(#KWDTYPE) TO(S)
ADD_ENTRY TO_LIST(#KWDSRCH)
***** Get the keywords from the string
USE BUILTIN(GET_KEYWORD_STRING) WITH_ARGS(#STRSRCH
    #KWDSRCH) TO_GET(#KWDFND #VALFND #STRLEFT)
***** Handle error
IF COND('#LEFTCOUNT > 0')
***** error processing
. . . . .
ELSE
***** Get the value for the file name keyword
GET_ENTRY NUMBER(1) FROM_LIST(#VALFND)
GET_ENTRY NUMBER(#KWDSTR) FROM_LIST(#VALFND)
CHANGE FIELD(#FILENAME) TO(#VALALPHA)
***** Use the file name
. . . . .
ENDIF

```

9.114 GET_LICENSE_STATUS

Retrieve the status of LANSA licenses in this LANSA system as at a particular date.

Running a regularly scheduled job using this Built-In Function can provide advance warning of a license about to expire.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Opt	License status as at Date (YYYYMMDD) If this argument is not provided or it is zero, today's date will be used.	8	8	0	0
2	A	Opt	Status of this license code is requested. If this argument is not provided, the status of all licenses as at argument1 date will be returned. (*)	3	3		
3	A	Opt	License Version (*) If this argument is not provided or is blank, a license version of "1" will be assumed.	1	1	0	0

4	A	Opt	Long License Code to check (*)	24	24	0	0
---	---	-----	--	----	----	---	---

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working List to contain the license statuses. (*)	*	*	0	0
2	A	Opt	Return code. OK = list returned partially or completely filled with license status. No more fields exist beyond those returned in the list. OV = list returned completely filled, but more entries than could fit in the list exist.	2	2		

(*) When License Version is set to "1", the 4th argument, if specified, is ignored. The calling RDML function must provide a working list with an aggregate entry length of exactly 4 bytes and with no more than 9999 entries. Each returned list entry is formatted as follows:

From	To	Description
1	3	License Code
4	4	Status Y/N. Y = License is valid for this machine as at the date used from argument 1.

When License Version is set to "2", the 2nd argument is ignored and the 4th

argument, if specified, is used as the license code whose status is requested. The calling RDML function must provide a working list with an aggregate entry length of exactly 25 bytes and with no more than 9999 entries. Each returned list entry is formatted as follows:

From	To	Description
1	24	License Code
25	25	Status Y/N. Y = License is valid for this machine as at the date used from argument 1.

Technical Notes

IBM i licenses: The license codes return will contain this LANSAs system's permanent licenses plus LANSAs Integrator key licenses where Integrator has been installed or upgraded using the IBM i installation processing. Where LANSAs Integrator has been installed separately, the Integrator licenses will not be returned by GET_LICENSE_STATUS.

Example

To find licenses which are currently valid but will expire in the next month.

```

DEFINE FIELD(#CODE1) TYPE(*CHAR) LENGTH(3)
DEFINE FIELD(#CODE2) TYPE(*CHAR) LENGTH(3)
DEFINE FIELD(#STATUS1) TYPE(*CHAR) LENGTH(1)
DEFINE FIELD(#STATUS2) TYPE(*CHAR) LENGTH(1)
DEF_LIST NAME(#WLIST1) FIELDS((#CODE1) (#STATUS1)) TYPE(*W
)
DEF_LIST NAME(#WLIST2) FIELDS((#CODE2) (#STATUS2)) TYPE(*W
)
DEF_LIST NAME(#BLIST) FIELDS((#CODE2) (#STATUS2))
DEFINE FIELD(#EXPDATE) TYPE(*DEC) LENGTH(8) DECIMALS(0)
DEFINE FIELD(#YESTERDAY) TYPE(*DEC) LENGTH(8) DECIMALS(0)

DEFINE FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)
***** Today + 31 days : to get licenses which will expire
***** next month

```

```

USE      BUILTIN(FINDDATE) WITH_ARGS(#YYYYMMDD 31 J J) TO_C
        XPCDATE)
***** WLIST2 will contain license status in 31 days time.
CLR_LIST #wlist2
USE      BUILTIN(GET_LICENSE_STATUS) WITH_ARGS(#EXPCDATE) T
        #WLIST2)
***** Today - 1 day : to get yesterday's date
USE      BUILTIN(FINDDATE) WITH_ARGS(#YYYYMMDD -1 J J) TO_C
        ESTERDAY)
***** WLIST1 will contain license status yesterday.
CLR_LIST #wlist1
USE      BUILTIN(GET_LICENSE_STATUS) WITH_ARGS(#YESTERDAY
        T(#WLIST1)
***** Compare the status in a month's time with the status
***** yesterday to find licenses which will expire in the
***** next month.
CLR_LIST #blist
SELECTLIST #wlist2
IF      COND('#status2 = N')
LOC_ENTRY IN_LIST(#WLIST1) WHERE('#code1 = #code2')
IF      COND('(#IO$STS = OK) *AND (#STATUS1 = Y)')
ADD_ENTRY #BLIST
ENDIF
ENDIF
ENDSELECT
DISPLAY FIELDS((#EXPCDATE)) BROWSELIST(#BLIST)

```

9.115 GET_LOGICAL_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of a physical files associated logical views and their descriptions from the data dictionary and returns them to the calling RDML function in a variable length working list.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	This Built-In Function is not supported if the X_RUN parameter DBII=*NONE.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Physical file name.	1	10		
2	A	Req	Physical file library.	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain logical file information. List must not be more than:	70	70		

			<p>32767 entries in Windows 9999 entries on IBM i.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 70 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description 1 - 10 Logical file 11 - 20 Logical file library 21 - 60 Description 61 - 70 <<future expansion>></p>				
2	A	Opt	<p>Return code.</p> <p>OK = list returned partially or completely filled with file details. No more logicals exist for this physical file.</p> <p>OV = list returned completely filled, but more files than could fit in the list exist. Typically used to indicate "more" fields in page at a time style list displays.</p> <p>NR = list was returned empty. Last file in the list is returned as blanks.</p> <p>ER = Physical file not found. Last file in the list is returned as blanks.</p>	2	2		

9.116 GET_MESSAGE

⇒ **Note:** Built-In Function Rules.

Gets the details of the next message from the program queue of the RDML function.

Normally the returned message details are then processed or printed by the RDML function in some non-standard way.

Messages on the program queue of an RDML function normally displayed on line 22/24 of the next screen presented to the user and then automatically cleared / removed.

Messages may have been placed on the program message queue by operating system commands, Built-In Functions, invalid I/O requests and/or RDML commands such as MESSAGE, VALUECHECK, etc.

Portability Considerations This BIF does not retrieve the Message File or Message Id number when running under LANSAs SuperServer. See the note following the example below.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Remove message from queue Y - message is removed N - message is not removed	1	1		

		Default value is Y.				
--	--	---------------------	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Message return code OK = message returned NO = no message found	2	2		
2	A	Opt	Message text	1	132		
3	A	Opt	Message number	1	7		
4	A	Opt	Message file name	1	10		
5	A	Opt	Message file library	1	10		
6	A	Opt	Message substitution variable	1	132		

Example

Insert a new name and address into file NAMES in a batch program. If an error is detected, print details of the name and address an exception report with **all** associated error messages.

```

DEFINE FIELD(#ERRTXT) TYPE(*CHAR) LENGTH(100)
      LABEL('Error :')
DEF_LINE NAME(#NAME) FIELDS(#CUSTNO #ADDRESS1
      #ADDRESS2 #ZIPCODE)
DEF_LINE NAME(#ERROR) FIELDS(#ERRTXT) IDENTIFY(*LABEL)

INSERT FIELDS(#NAME) TO_FILE(NAMES) VAL_ERROR(*NEXT)
IF_STATUS IS_NOT(*OKAY)
      PRINT LINE(#NAME)
      USE BUILTIN(GET_MESSAGE) TO_GET(#RETCODE #ERRTXT)

```

```
DOWHILE COND('#RETCODE = OK')
  PRINT LINE(#ERROR)
  USE BUILTIN(GET_MESSAGE)
    TO_GET(#RETCODE #ERRTXT)
  ENDWHILE
ENDIF
```

Note:

Running the same program under IBM i and in LANSa SuperServer mode will produce different messages.

A retrieved message using this BIF on the IBM i will display:

Message text: Record to be updated has been changed by another job/user
Message number: IOM0017
Message file name: DC@M01

Running in LANSa SuperServer mode (against the IBM i) will display:

Message text: Record to be updated has been changed by another job/user
Message number: *STCMMSG
Message file name: 2

This is a design consideration that, due to technical complexities, will not be changed in the short term.

9.117 GET_MESSAGE_DESC

⇒ **Note:** Built-In Function Rules.

Gets the description associated with a message number in a message file. Normally the returned message details are then processed or printed by the RDML function.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Message number	1	7		
2	A	Req	Message file	1	10		
3	A	Opt	Message file library	1	10		
4	A	Opt	Message substitution value(s) Multiple substitution values can be used. Substring values together to match message definition. For example, "Function &1 in &2 failed", with substitution values *CHAR7 and *CHAR10 provided. The second value must begin in position 8.	1	132		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Message description	1	132		

Example

Execute a subroutine to print an error / exception report. The subroutine is passed an error message number.

```

DEFINE FIELD(#ERRTXT) TYPE(*CHAR) LENGTH(132)
      LABEL('Error :')
DEFINE FIELD(#TEXT) TYPE(*CHAR) LENGTH(132)
DEFINE FIELD(#MSGID) TYPE(*CHAR) LENGTH(7)
DEFINE FIELD(#MSGT) TYPE(*CHAR) LENGTH(132)
DEF_LINE NAME(#NAME) FIELDS(#CUSTNO #ADDRESS1 #ADDRESS2)
DEF_LINE NAME(#ERROR) FIELDS(#ERRTXT) IDENTIFY(*LABEL)

' ' Some processing ' '
' ' ' '

INSERT FIELDS(#NAME) TO_FILE(NAMES) VAL_ERROR(*NEXT)
IF_STATUS IS_NOT(*OKAY)
PRINT LINE(#NAME)
CHANGE FIELD(#TEXT) TO('xxxxxxx')
EXECUTE SUBROUTINE(ERRPRT) WITH_PARMs(ERR0003 #TEXT)
ENDIF
' ' More processing ' '
' ' ' '

SUBROUTINE NAME(ERRPRT) PARMs(#MSGID)
USE BUILTIN(GET_MESSAGE_DESC) WITH_ARGS(#MSGID #ERROR)
   ""*LIBL"" #TEXT) TO_GET(#MSGT)
PRINT LINE(#ERROR)

ENDROUTINE

```

9.118 GET_MESSAGE_LIST

⇒ **Note:** Built-In Function Rules.

The only required parameter will be a message file. This will load the list with each subsequent message file / message stored in the message table LX_MSG. If a language other than *ALL is specified only messages with the specified language will be returned.

If a message id is specified, all messages following the message id will be returned to the list. (NB a language must be specified if a message id is specified.)

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Search Message File	1	10		
2	A	Opt	Language Code If the special value *ALL is specified all languages will be returned, otherwise only messages for the selected language will be listed.	1	4		
3	A	Opt	Message Identifier	1	7		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing message details. List must not be more than 32767 entries. Message details to be formatted as follows: From - To Description 1 - 4 Language 5 - 14 Message File 15 - 21 Message Identifier 22 - 153 Message Text	153	153		
2	A	Req	The last Language in the returned list.	1	4		
3	A	Req	The last Message File in the returned list.	1	10		
4	A	Req	The last Message Identifier in the returned list.	1	7		
5	A	Req	Return Code OK = list returned partially or completely filled with message details. No more Messages exist beyond those returned in the list OV = list returned completely filled, but more Messages than could fit in the list still exist NR = list was returned empty. Last Message File/ Message Identifier in list returned as blanks	2	2		

9.119 GET_ML_VARIABLE

⇒ **Note:** All Multilingual Built-In Functions in [Built-In Function Rules](#).

Retrieves a multilingual variable definition.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Multilingual variable name	5	20		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code (OK, ER)	2	2		
2	N	Req	Length / Total digits	3	3	0	0
3	L	Req	Working list to contain multilingual definition information. List must not be more than: 32767 entries in Windows	82	82		

9999 entries on IBM i.

RDML

An RDML list must be formatted with an aggregate entry length of exactly 82 bytes.

Bytes 1-4: Language code

Bytes 5-82: Multilingual variable value.

RDMLX

An RDMLX list must be formatted as:

Alpha (4): Language code

NChar (39): Multilingual variable value.

9.120 GET_MULTVAR_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of multilingual variables (*MTXT) and their value in the current language and returns them to the calling RDML function in a variable length working list.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Positioning *MTXT variable. The returned list starts with the first *MTXT variable whose name is greater than the value passed in this argument.	1	20		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain *MTXT variable information. List must not be more than:	108	108		

			<p>32767 entries in Windows 9999 entries on IBM i.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 108 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description 1 - 20 *MTXT variable name 21 - 98 *MTXT value in current language 99 - 108 << for future expansion >></p>				
2	A	Opt	Last *MTXT variable in list Typically this value is used as the positioning argument on subsequent calls to this Built-In Function.	1	20		
3	A	Opt	<p>Return code.</p> <p>OK = list returned partially or completely filled with *MTXT variable details. No more *MTXT variables exist beyond those returned in the list.</p> <p>OV = list returned completely filled, but more *MTXT variables than could fit in the list exist. Typically used to indicate "more" *MTXT variables in page at a time style list displays.</p> <p>NR = list was returned empty. Last *MTXT variable in the list is returned as blanks.</p>	2	2		

Example

A user wants to print a list of all *MTXT variables.

```

DEF_LIST    NAME(#MTXLST) FIELDS(#MTXNAM #MTXVAL #SPARE
            TYPE(*WORKING) ENTRYS(1000)
*****      -Define the report layout-
DEF_REPORT  PRT_FILE(QSPRT)

```

```
DEF_HEAD    NAME(#HEAD01) FIELDS(#TEXT #PAGE . . . )
DEF_LINE    NAME(#MTXPRT) FIELDS(#MTXNAM #MTXVAL)
*****      -Set start *MTXT variable to blanks-
CHANGE      FIELD(#MTXVAR) TO(*BLANKS)
*****      -Get list of system variables-
USE          BUILTIN(GET_MULTVAR_LIST) WITH_ARGS(#MTXVAR)
            TO_GET(#MTXLST)
*****      -Process list-
SELECTLIST  NAME(#MTXLST)
*****      -Print *MTXT variables-
PRINT       LINE(#MTXPRT)
ENDSELECT
*****      -Close printer file-
ENDPRINT
```

9.121 GET_NUM_AREA

⇒ **Note:** Built-In Function Rules.

Gets a numeric value from a numeric data area.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Data area name	1	10		
2	A	Opt	Library name Default: *LIBL	1	10		
3	A	Opt	Lock data area Y - lock data area. N - do not lock data area. Default: N	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Opt	Returned value		15	0	0

Example

Retrieve a batch number #BATCH from data area named NEXTBATCH which should be located via the job's library list.

Increment the batch number value and place the incremented value back into the data area.

Make sure that no 2 jobs can be assigned the same batch number by using the lock and unlock options.

```
USE    BUILTIN(GET_NUM_AREA)
      WITH_ARGS(NEXTBATCH "*"LIBL" 'Y') TO_GET(#BATCH)
CHANGE FIELD(#BATCH) TO('#BATCH + 1')
USE    BUILTIN(PUT_NUM_AREA)
      WITH_ARGS(#BATCH NEXTBATCH "*"LIBL" 'Y')
```

9.122 GET_PHYSICAL_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of physical files and their descriptions from the data dictionary and returns them to calling RDML function in a variable length working list.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	This Built-In Function is not supported if the X_RUN parameter DBII=*NONE.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Positioning file value. The returned list starts with the first file from the dictionary whose name is greater than the value passed in this argument.	1	10		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain File information. List must not be more than:	70	70		

			<p>32767 entries in Windows 9999 entries on IBM i.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 70 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 10 Physical file name</p> <p>11 - 20 Physical file library</p> <p>21 - 60 Description</p> <p>61 - 63 On IBM i are set to blank.</p> <p>On Windows, they contain this Visual LANSA-specific information:</p> <p>61 - 61 File type: N = LANSA File Y = Other File (loaded on IBM i) P = Other File (loaded on Windows)</p> <p>62 - 62 Automatic RRNO (Y or N)</p> <p>63 - 63 @@RRNO & @@UPID on file (Y or N)</p> <p>64 - 64 RDMLX file (Y or N)</p> <p>65 - 70 <<future expansion>></p>				
2	A	Opt	Last file in returned list Typically this value is used as the positioning argument on subsequent calls to this Built-In Function.	1	10		
3	A	Opt	<p>Return code.</p> <p>OK = list returned partially or completely filled with file details. No more files exist beyond those returned in the list.</p> <p>OV = list returned completely filled, but more files than could fit in the list exist. Typically used to indicate "more" files in page at a time style list displays.</p> <p>NR = list was returned empty. Last file in the list is returned as blanks.</p>	2	2		

Example

This function could be used to write a program that allows a site to modify an existing LANSAs database.

```
DEF_LIST  NAME(#FILLST) FIELDS(#FILNAM #FILLIB #FILDES #SP
          TYPE(*WORKING) ENTRYS(10)
DEF_LIST  NAME(#FILDSP) FIELDS((#SELECTOR *SEL) #FILNAM #
          #FILDES)
***** -Clear lists-
CLR_LIST  NAMED(#FILLST)
CLR_LIST  NAMED(#FILDSP)
***** -Request file to start from in list-
REQUEST   FIELDS(#STRNFL) TEXT(('File to start from' 5 5))
***** -Get the list of files-
USE       BUILTIN(GET_PHYSICAL_LIST) WITH_ARGS(#STRNFL)
          TO_GET(#FILLST #LAST #RETCOD)
***** -If records found-
IF        COND('( #RETCOD *EQ OK) *OR ( #RETCOD *EQ OV)')
SELECTLIST NAMED(#FILLST)
ADD_ENTRY  TO_LIST(#FILDSP)
ENDSELECT
*****
DISPLAY   BROWSELIST(#FILDSP)
***** -Process selected records-
SELECTLIST NAMED(#FILDSP) GET_ENTRYS(*SELECT)
EXECUTE   SUBROUTINE(FILE_EDIT)
ENDSELECT
ELSE
MESSAGE   MSGTXT('No files found .... Program ended')
RETURN
ENDIF
```

9.123 GET_PROCESS_ATTR

⇒ **Note:** Built-In Function Rules.

Gets attributes of a process definition, that is being edited within an edit session, previously started using the START_PROCESS_EDIT Built-In Function.

Attributes set or returned by this Built-In Function have the same editing and validation rules as the equivalent online facility provided in a full LANSAs development environment.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a commercial application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of attribute to get Valid attribute names are: DESC- Process Description TYPE- Process Type OPTCOM - Optimize for remote comms ENAWEB - Enable for the Web	1	50		

		ENAXML - Enable for XML Generation TOTFUN - Total associated functions EXISTS - Checks for existence of function name specified in bytes 7 to 13 of argument (directly following the EXISTS string in bytes 1 to 6).				
--	--	--	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		
2	A	Req	Returned process attribute For attribute DESC: The process description up to 40 characters in length. For attribute TYPE: SAA/CUA ACT/BAR For attribute OPTCOMM: Y – Optimized for remote comms N – Not optimized for remote comms For attribute ENAWEB: Y – Enabled for the Web N – Not enabled for the Web For attribute ENAXML: Y – Enabled for XML Generation N – Not enabled for XML Generation For attribute TOTFUN: Character 3 value containing a number in the range 000 - 990.	1	256		

		For attribute EXISTS: Y – Function name specified exists in the process N – Function name specified does not exist in the process				
--	--	---	--	--	--	--

9.124 GET_PROCESS_INFO

⇒ **Note:** Built-In Function Rules.

Retrieves a list of process related information from the LANSAs internal database and returns it to the calling RDML function in a variable length, working list.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Process name.	1	10		
2	A	Req	Type of process related information to retrieve. Valid types are: PROCATTACH - Attached processes/ functions MLATTR- Multilingual attributes	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return code</p> <p>OK = list returned partially or completely filled. No more of this type of information exists for this process.</p> <p>OV = list returned completely filled, but more of this type of information than could fit in the list exists.</p> <p>NR = list was returned empty. Last entry in the list is returned as null.</p> <p>ER = Process not found. Last entry in the list is returned as null.</p>	2	2		
2	L	Req	<p>Working list to contain process related information.</p> <p>List must not be more than: 32767 entries in Windows 9999 entries on IBM i.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 100 bytes.</p> <p>For type PROCATTACH: Each returned list entry is formatted as follows:</p> <p>From - To Description 1 - 10 Attached process name 11 - 17 Attached function name (*ALL if process is attached) 18 - 100 <<future expansion>></p> <p>For type MLATTR: Each returned list entry is formatted as follows:</p>	100	100		

		1 - 4 Language code				
		5 - 44 Process description				
		45 - 100 <<future expansion>>				

9.125 GET_PROCESS_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of processes and their descriptions from the LANSAs internal database and returns them to the calling RDML function in a variable length, working list

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Positioning process value. The returned list starts with the first process from the dictionary whose name is greater than the value passed in this argument.	1	10		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain Process information. List must not be more than: 32767 entries in Windows	60	60		

			<p>9999 entries on IBM i.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 60 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description 1 - 10 Process name 11 - 50 Description 51 - 60 <<future expansion>></p>				
2	A	Opt	Last process in returned list Typically this value is used as the positioning argument on a subsequent calls to this Built-In Function.	1	10		
3	A	Opt	<p>Return code.</p> <p>OK = list returned partially or completely filled with process details. No more processes exist beyond those returned in the list.</p> <p>OV = list returned completely filled, but more processes than could fit in the list exist. Typically used to indicate "more" processes in page at a time style list displays.</p> <p>NR = list was returned empty. Last process in the list is returned as blanks.</p>	2	2		

Example

A program can be created, using this function, to compile a series of processes in an overnight job.

```

FUNCTION OPTIONS(*DIRECT)
DEFINE FIELD(#STARTPRC) REFFLD(#PROCESS) DESC('Start Search wi
DEFINE FIELD(#LASTPRC) REFFLD(#PROCESS) DESC('Last retrieved:')
DEFINE FIELD(#SPARE) REFFLD(#PROCESS)
OVERRIDE FIELD(#STD_INSTR) COLHDG('Name (Description)')
OVERRIDE FIELD(#STD_CMPAR) DESC('Return Code (OV,OK or NR)')

```

```

DEF_LIST NAME(#PRCLST) FIELDS(#PROCESS #PARTDESC #SPARE) '
DEF_LIST NAME(#BRWLST) FIELDS((#STD_INSTR *NOID))
*
CHANGE FIELD(#LASTPRC #STD_CMPAR) TO(*BLANKS)
*
BEGIN_LOOP
REQUEST FIELDS(#STARTPRC #LASTPRC (#STD_CMPAR *OUT)) IDEI
IF COND('#lastprc *ne #blanks')
CHANGE FIELD(#STARTPRC) TO(#lastprc)
ENDIF
CLR_LIST NAMED(#PRCLST)
USE BUILTIN(GET_PROCESS_LIST) WITH_ARGS(#STARTPRC) TO_GE
CASE OF_FIELD(#STD_CMPAR)
WHEN VALUE_IS('= OV')
CLR_LIST NAMED(#BRWLST)
SELECTLIST NAMED(#PRCLST)
USE BUILTIN(BCONCAT) WITH_ARGS(#PROCESS '(' #PARTDESC ')') TO
ADD_ENTRY TO_LIST(#BRWLST)
ENDSELECT
WHEN VALUE_IS('= OK')
MESSAGE MSGTXT('No more matching process names')
CHANGE FIELD(#LASTPRC) TO(#BLANKS)
WHEN VALUE_IS('= ER')
MESSAGE MSGTXT('No process names matching search')
ENDCASE
END_LOOP

```


1	A	R	<p>Return Code:</p> <p>OK – Upgrade Code found</p> <p>NR – Upgrade Code not known.</p> <p>VE – Validation Error. A list format is incorrect. For example, Too few columns or incorrect column type.</p> <p>OV – more attributes available than size of list. Extend list to obtain other attributes.</p> <p>ER – Error. Could not complete request.</p>	2	2		
2	List	R	<p>Product Attributes</p> <p>Only contains entries if return code is OK or OV.</p> <p>All attributes are returned. If an attribute is not valid in the current context then the Attribute Validity is set to ER and the Attribute Value is set to the API Error Code. Refer to MsiGetProductInfoEx() error codes for error code reason.</p> <p>For example, 1608 is ERROR_UNKNOWN_PROPERTY. The values can be looked up in winerror.h. Note that the error codes are all negated. Remove the negative sign to locate the meaning.</p> <p>List structure is flexible with only these requirements:</p> <p>NCHAR/NVARCHAR fields preferably with *LC attribute</p> <p>Field 1, NCHAR or NVARCHAR =Attribute Id;</p> <p>Field 2, NCHAR or NVARCHAR =Attribute Value;</p> <p>Field 3, Alpha(2)=Attribute Validity:</p>	1	2147483647		

			OK, ER = value not available, OV = Attribute Value length is too short for the value.				
3	List	O	<p>Patch Attributes</p> <p>If the Patch List is provided then Patch Attributes are listed.</p> <p>Only contains entries if Return Code is OK or OV.</p> <p>All attributes are returned. If an attribute is not valid in the current context then the Attribute Validity is set to ER. And the Attribute Value is set to the API Error Code. Refer to MsiGetPatchInfoEx () error codes e.g. -1608 is ERROR_UNKNOWN_PROPERTY. The values can be looked up in winerror.h. Note that the error codes are all negated. Remove the negative sign to locate the meaning.</p> <p>List structure is flexible with only these requirements:</p> <p>NCHAR/NVARCHAR fields preferably with *LC attribute</p> <p>Field 1, NCHAR or NVARCHAR =Attribute Id;</p> <p>Field 2, NCHAR or NVARCHAR =Attribute Value;</p> <p>Field 3, Alpha(2)=Attribute Validity: OK=valid, ER = invalid , OV = Attribute Value length is too short for the value.</p>	1	2147483647		

Notes

This Built-In Function lists all products that are related to the Upgrade Code. The current Windows Installer documentation implies that there should only be one, but the code will list all products that are returned.

Products installed across all users in the system are listed. You may not have access rights to the Upgrade Code you have specified.

Attributes may be added to the list, so do not expect them in any particular order or even that the one you are looking for is returned. If the code lists a value, it will at least set the attribute return code to ER, it will not omit it from the list.

References are made below to attributes that are defined winerror.h and msi.h which are C header files.

Product Attributes

UpgradeCode - 38 byte GUID (Input value, but as it is optional the actual value used is returned in the list).

For each Product related to the Upgrade Code the following are returned:

ProductCode – 38 byte GUID

InstallContext – a value in the enumeration MSIINSTALLCONTEXT

SID – Security Identifier of the account under which this product instance exists.

The following values are easily mapped to the Attribute Identifier returned. For example, INSTALLPROPERTY_INSTALLEDPRODUCTNAME returns

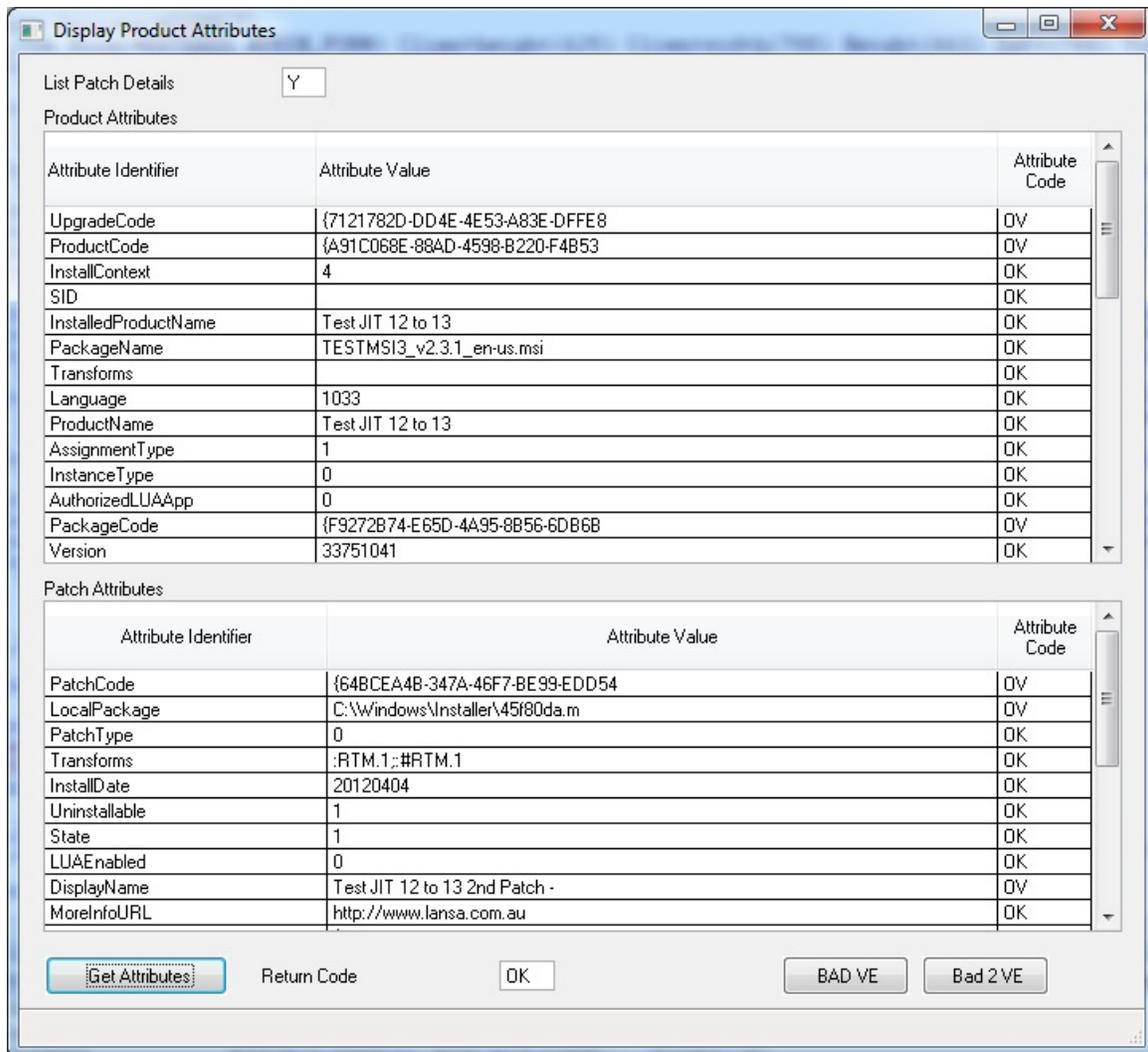
InstalledProductName. Listing the value used by C will enable you to search the Web for what the attribute means.

```
INSTALLPROPERTY_INSTALLEDPRODUCTNAME,  
INSTALLPROPERTY_PACKAGENAME      ,  
INSTALLPROPERTY_TRANSFORMS       ,  
INSTALLPROPERTY_LANGUAGE         ,  
INSTALLPROPERTY_PRODUCTNAME      ,  
INSTALLPROPERTY_ASSIGNMENTTYPE   ,  
INSTALLPROPERTY_INSTANCETYPE     ,  
INSTALLPROPERTY_AUTHORIZED_LUA_APP ,  
INSTALLPROPERTY_PACKAGECODE      ,  
INSTALLPROPERTY_VERSION          ,  
INSTALLPROPERTY_PRODUCTICON      ,
```

```
// Product info attributes: installed information
```

INSTALLPROPERTY_INSTALLEDPRODUCTNAME ,
INSTALLPROPERTY_VERSIONSTRING ,
INSTALLPROPERTY_HELPLINK ,
INSTALLPROPERTY_HELPTELEPHONE ,
INSTALLPROPERTY_INSTALLLOCATION ,
INSTALLPROPERTY_INSTALLSOURCE ,
INSTALLPROPERTY_INSTALLDATE ,
INSTALLPROPERTY_PUBLISHER ,
INSTALLPROPERTY_LOCALPACKAGE ,
INSTALLPROPERTY_URLINFOABOUT ,
INSTALLPROPERTY_URLUPDATEINFO ,
INSTALLPROPERTY_VERSIONMINOR ,
INSTALLPROPERTY_VERSIONMAJOR ,
INSTALLPROPERTY_PRODUCTID ,
INSTALLPROPERTY_REGCOMPANY ,
INSTALLPROPERTY_REGOWNER ,
INSTALLPROPERTY_INSTALLEDLANGUAGE ,
INSTALLPROPERTY_PRODUCTSTATE ,
INSTALLPROPERTY_LASTUSEDSOURCE ,
INSTALLPROPERTY_LASTUSEDTYPE ,
INSTALLPROPERTY_MEDIAPACKAGEPATH ,
INSTALLPROPERTY_DISKPROMPT,

For example, following is a list of a few of the attributes returned for an example Product. Note that the Attribute Value field length was only 30 so a number of attributes, such as **UpgradeCode**, have a status of OV. LANSAs has implemented the standard behaviour of truncating the value:



Patch Attributes

Each of the attributes will be listed starting with the Patch Code. When a Patch Code entry repeats, its the start of another patch. There can be many patches in some products, such as the Microsoft Windows product itself, so it's advisable to use a dynamic list.

PatchCode – 38 byte GUID

INSTALLPROPERTY_LOCALPACKAGE,
 INSTALLPROPERTY_PATCHTYPE,
 INSTALLPROPERTY_TRANSFORMS,
 INSTALLPROPERTY_INSTALLDATE,
 INSTALLPROPERTY_UNINSTALLABLE,
 INSTALLPROPERTY_PATCHSTATE,
 INSTALLPROPERTY_LUAENABLED,
 INSTALLPROPERTY_DISPLAYNAME,

INSTALLPROPERTY_MOREINFOURL

9.127 GET_PROPERTIES

⇒ **Note:** Built-In Function Rules.

Returns the version details of a single LANSa OBJECT.

For use with

LANSa for i	NO
Visual LANSa for Windows	YES
Visual LANSa for Linux	YES

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object Name. Refer to Type of Object Name for format of the name entered here.	1	20		
2	A	Opt	Type of Object Name F = Full. This is the default. The full library name has been supplied e.g. for Linux Object Name = "liblp_dpb.so" for the object lp_dpb P = Partial. Only the base library name has been supplied. The platform-specific data must be added. E.g. Object Name = "X_PDF" which for Windows would be converted to "X_PDF.DLL" and for Linux would be converted to "libx_pdf.so"	1	1		
3	A	Opt	Partition or System Object P = Partition. This is the default. The object will be located in the execute directory for the partition being processed e.g.	1	1		

		<p>C:\X_WIN95\X_LANSA\X_DEM\EXECUTE</p> <p>S = System. The object will be located in the system execute directory e.g.</p> <p>C:\X_WIN95\X_LANSA\EXECUTE</p>				
--	--	--	--	--	--	--

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	<p>Identification list for an object</p> <p>From - To Description</p> <p>1 - 2 Identifier Number</p> <p>3 - 4 Option Number</p> <p>5 - 44 Data</p> <p>See values below.</p>	44	44		
2	A	Req	<p>Return Code</p> <p>OK = The list was returned without error</p> <p>NR = List was returned empty. This is usually due to either a non-LANSA object being queried or when the LANSAs object was generated before the information was made available.</p> <p>ER = An Error occurred during the getting of the list.</p>	2	2		

Note: Identifier, Option Number and Build Number (in Data description below) are zero filled and right justified.

Identifier Number

00 = Object Type

- 01 = Object Name
- 02 = Object Extension
- 03 = DLL Name
- 04 = Build Date
- 05 = Build Time
- 06 = Supports Components
- 07 = Collection Name
- 08 = Visual LANSAs Internal Identification

Option Number

This only applies when the Identifier Number is 08.

- 00 = Build Number
- 01 = Release Number
- 02 = Build Date
- 03 = Copyright Dates

Data

Object Type: FD = File
PF = Function
PD = Process
DF = Component
XX = LANSAs Internal Object

Object
Name

Object
Extension

DLL Name

Build Date * mmm dd yyyy

Build Time hh:mm:ss

*

Component Y=Supports Components
Support N=Does not support Components

Not used for Functions.

Collection Name This name is the same as the Library Name, except that it is truncated when the Library Name is too long. For example, a file in the LANSAs demonstration system would have a value of XDEMOLIB.

Not used for a Function.

Visual LANSAs Internal Option Number 00 = Build Number
01 = Release Number
02 = Build Date
03 = Copyright Dates

* All dates and times are in the language format of the computer on which the object was built. Therefore, LANSAs internal objects, such as X_PDF.DLL, will always return an English style date because they were built in Australia. On the other hand, for an object built in France such as MYFUNC.DLL, the date will be French.

Times do not contain any time zone information so it is not possible to convert the time into local time for comparison purposes. The times should only be used for comparison when you are certain that all times you are comparing were generated in the same time zone.

Examples

Following are two examples of the GET_PROPERTIES Return Values. One shows the values for a Function, the other is for a File:

Identifier Number	Option Number	Function Example	File Example
00	Spaces	PF	FD
01	Spaces	MYFUNC	LX_F02
02	Spaces	MYPROC	LX_DTA
03	Spaces	MYFUNC	LX_F02
04	Spaces	Oct 12 2001	Jul 16 1999
05	Spaces	12:53:40	15:15:07

06	Spaces	Spaces. Does not apply to a Function.	Y
07	Spaces	Spaces. Does not apply to a Function.	LX_DTA
08	01	9.1	7.8
08	02	0000002208	0000000000
08	03	Oct 10 2001	Jun 24 1999
08	04	1993-2001	1993-1999

9.128 GET_REGISTRY_VALUE

⇒ **Note:** Built-In Function Rules.

Processing

Returns the Value for the specified Registry Key.

When the length of an Argument is stated as being greater than 50, this is only true for Fields. Literal values are restricted to a maximum length of 50. This is especially true for the three arguments in this BIF. All the arguments are limited to a length of 50 unless a field is used.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No.	Type	Req/Opt	Description	Min Len	Max Len
1	A	R	Registry Root e.g. HKEY_CLASSES_ROOT,HKEY_LOCAL_MACHINE	1	256
2	u	R	Registry Path e.g. WinZip\shell\open\command	1	256
3	u	O	Registry Key Name If not specified the (Default) value for the specified path will be returned, otherwise specify the name of the key.	1	256
4	N	O	Registry Hive to use: 32 or 64 Any other value will use the default for the application.	1	4

		That is, a 32 bit application will write to wow6432 while a 64 bit application will write to wow6464. This argument is ignored on a 32 bit operating system.	
--	--	---	--

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	X	R	Registry Key Value Refer to Key Value Note for details.	1	Unlimited		Unlimited
2	A	R	Return Code OK – Key found and Value Returned ER – Key could not be found	2	2		
3	A	O	Value Type S – String B – Binary D – DWORD X – Expanded string. Can contain environment variables which will be expanded on return from this Built-In Function.	1	1		

Key Value Note

If a field value is saved by PUT_REGISTRY_VALUE, a field with the same type must be used to get the correct value by GET_REGISTRY_VALUE.

The table shows the supported field types for each key type.

Key Type	Supported Field Types

S & X	Alpha, BLOB, Boolean, Char, CLOB, Date, DateTime, Float, Integer, Packed, Signed, Time and Char.
B	Binary, VarBinary, Alpha, Char and String.
D	Alpha, Char, String and less than 8 bytes Integer.

Example

Values PUT and the returned value on a GET from the Registry.

Data Type	PUT	GET
S	ABC1234	ABC1234
D	7000	7000
D	9999999999	0 (the biggest number for DWORD is 4294967295)
D	-12	-12
B	AAAAA	AA AA 0A
B	WEWE	0E 0E (because 'W' is not a HEX number)

9.129 GET_SESSION_VALUE

Returns the value for a specified X_RUN parameter.

The following parameters are supported :

APPL, ASCT, ASLU, ASPW, ASSQ, ASST, ASTC, CDLL, CIPH, CMTH, DBID, DBUS, DBUT, DBII, DBIT, DELI, DPTH, EXPS, GUSR, HELP, HLPF, HMJB, INST, ITRO, ITRL, ITRM, ITRC, LANG, ODBI, PKGD, PPTH, PSCT, PSTC, PSLU, PSST, PSPW, QUET, RPTH, SUBD, UDEF, USER, USEX, WDTM plus all [Windows Printing Extension](#) parameters such as WPEN, WPPN, WPPS, WPPD, WPDF, WPDS, WPFO, WPAS, XCMD (only becomes active for jobs submitted after it has been set).

LOB Temporary File Parameters

The following pseudo X_RUN parameters are supported for LOB temporary file handling. They return directories that can be used in code to temporarily store LOB files for the current job or user. Use the [OV_FILE_SERVICE](#) Built-In Function to ensure the directory exists before creating files:

LTPP	Temporary LOB Path for the current job. Automatically cleaned up when LANSAs completes execution.
LTPU	Temporary LOB Path for the current user. Not cleaned up when LANSAs completes execution.

Pseudo X_RUN parameters are supported by Visual LANSAs for Windows only:

*ENC	Encoded root path. Returns an alphanumeric string that matches the Windows registry key for the currently executing application.
*HST	Returns the Host Monitor status (the meaning of the return values is not currently published).
*LPC	Returns 'Y' if the LANSAs development environment is running. Otherwise returns 'N'.
*MEX	Returns the maximum export version that can be imported into the system, as 3 characters with a leading zero. For example, '001'.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	X_RUN Parameter Name Note: USER vs USEX USER always uses value in upper case USEX maintains case exactly as entered. For example: If AbCd is entered, USER = ABCD USEX = AbCd	1	4		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	X_RUN Parameter Value	1	256		
2	A	Req	Return Code OK = Parameter Returned OK ER = Error returning value	2	2		

9.130 GET_SPLF_LIST_ENTRY

⇒ **Note:** Built-In Function Rules.

This Built-In Function is used in conjunction with START_RTV_SPLF_LIST and END_RTV_SPLF_LIST. The START_RTV_SPLF_LIST must be used first to provide the selection criteria for the retrieval of spool files. The selection criteria which can be specified are User Name, Output queue name and library, Form Type, User Data and Status. Once the START_RTV_SPLF_LIST has been used to establish the selection, this Built-In Function, GET_SPLF_LIST_ENTRY can be used to retrieve the details of the spool files. The END_RTV_SPLF_LIST must be used after the list of spool files have been retrieved. This will close the list and release the storage allocated to that list.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Starting record number. This must be 1 for the first use of this Built-In Function after each START_RTV_SPLF_LIST. On subsequent calls it will normally be one more than the last record returned on the previous call. This value must be a positive number greater than zero.	5	15	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	<p>Last record number returned.</p> <p>This will return the number of the last record returned in this list. This can be used to establish the starting record number for the next use of this function. The starting record for the next use should be this number plus one.</p>	5	15	0	0
2	L	Req	<p>Working list to contain details of the spool files retrieved. The calling RDML function must provide a working list with an aggregate entry length of exactly 160 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description</p> <p>1 - 10 A(10) Spool file Name</p> <p>11 - 20 A(10) Job Name</p> <p>21 - 30 A(10) User</p> <p>31 - 36 A(6) Job Number</p> <p>37 - 40 A(4) Spool file Number</p> <p>41 - 44 P(7,0) Total Pages</p> <p>45 - 48 P(7,0) Current Page</p> <p>49 - 52 P(7,0) Copies left to print</p> <p>53 - 62 A(10) Output Queue name</p> <p>63 - 72 A(10) Output Queue Library</p> <p>73 - 82 A(10) User Data</p> <p>83 - 92 A(10) Status</p> <p>93 - 102 A(10) Form Type</p> <p>103 - 104 A(2) Priority</p> <p>105 - 136 A(32) Reserved</p> <p>137 - 146 A(10) Device type</p> <p>147 - 160 A(14) Reserved</p>	160	160		
3	A	Opt	Return code	2	2		

		<p>OK = list returned partially or completely filled. No more spool files exist for the selection.</p> <p>OV = list returned completely filled, and more spool files exist.</p> <p>NR = list was returned empty.</p> <p>ER = error encountered in retrieval of spool files. Starting record may be invalid.</p>				
--	--	---	--	--	--	--

Example

A user wants to retrieve all the spool files on output queue PAYOUTQ in library PAYLIB and then perform some processing on each spool file.

***** Define arguments and lists

```
DEFINE FIELD(#RETURN) TYPE(*CHAR) LENGTH(2)
```

```
DEFINE FIELD(#SPLF) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#JOB) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#USER) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#JOBNO) TYPE(*CHAR) LENGTH(6)
```

```
DEFINE FIELD(#SPLFNO) TYPE(*CHAR) LENGTH(4)
```

```
DEFINE FIELD(#TOTPAGE) TYPE(*DEC) LENGTH(7) DECIMALS(0)
```

```
DEFINE FIELD(#CURPAGE) TYPE(*DEC) LENGTH(7) DECIMALS(0)
```

```
DEFINE FIELD(#COPIES) TYPE(*DEC) LENGTH(7) DECIMALS(0)
```

```
DEFINE FIELD(#OUTQ) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#OUTQL) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#USERDATA) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#STATUS) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#FORM) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#PRTY) TYPE(*CHAR) LENGTH(2)
```

```
DEFINE FIELD(#RESV1) TYPE(*CHAR) LENGTH(32)
```

```
DEFINE FIELD(#DEVICE) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE FIELD(#RESV2) TYPE(*CHAR) LENGTH(14)
```

```
DEFINE FIELD(#START) TYPE(*DEC) LENGTH(7) DECIMALS(0) DEI
```

```
DEFINE FIELD(#LASTREC) TYPE(*DEC) LENGTH(7) DECIMALS(0) I
```

```

*****
DEF_LIST  NAME(#LIST) FIELDS((#SPLF) (#JOB) (#USER) (#JOBNO) (#
*****
***** Retrieve spool files on output Q PAYLIB/PAYOUTQ
*****
USE      BUILTIN(START_RTV_SPLF_LIST) WITH_ARGS('*ALL' PAYC
IF      COND('#RETURN = OK')
*****
CLR_LIST  NAMED(#LIST)
BEGIN_LOOP
USE      BUILTIN(GET_SPLF_LIST_ENTRY) WITH_ARGS(#START) TO_
IF      COND('(#return = ER) *OR (#return = NR)')
LEAVE
ENDIF
SELECTLIST NAMED(#LIST)
*****
***** At this point some processing on the spool file can
***** be done.
***** eg IBM i commands such as CPYSPLF DLTSPFL
***** or release (RLSSPLF) all files which are currently
***** held (have a status of *HELD)
*****
ENDSELECT
IF      COND('(#return = OV)')
CHANGE  FIELD(#START) TO('#LASTREC + 1')
ELSE
LEAVE
ENDIF
END_LOOP
*****
USE      BUILTIN(END_RTV_SPLF_LIST)
ENDIF

```

9.131 GET_SYSTEM_VARIABLE

⇒ **Note:** Built-In Function Rules.

Retrieves a system variable definition.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	System variable name	5	20		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code (OK, ER)	2	2		
2	A	Opt	Description	1	40		
3	A	Opt	STATIC or DYNAMIC	7	7		
4	A	Opt	Data type (ALPHA, NUMBER)	6	6		
5	N	Opt	Length / Total digits	3	3	0	0

6	N	Opt	Decimal positions	1	1	0	0
7	A	Opt	Evaluation program	10	10		
8	A	Opt	Ev. program type (FUN, 3GL)	3	3		

9.132 GET_SYSVAR_LIST

⇒ **Note:** Built-In Function Rules.

Retrieves a list of system variables, their descriptions, programs and program types and returns them to the calling RDML function in a variable length working list.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Positioning system variable. The returned list starts with the first system variable whose name is greater than the value passed in this argument.	1	20		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain system variable information. List must not be more than:	80	80		

			<p>32767 entries in Windows 9999 entries on IBM i.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 80 bytes.</p> <p>Each returned list entry is formatted as follows:</p> <p>From - To Description 1 - 20 System variable name 21 - 60 System variable description 61 - 70 System variable program 71 - 73 Program type 74 - 80 << for future expansion >></p>				
2	A	Opt	Last system variable in list. Typically this value is used as the positioning argument on subsequent calls to this Built-In Function.	1	20		
3	A	Opt	<p>Return code.</p> <p>OK = list returned partially or completely filled with system variable details. No more system variables exist beyond those returned in the list.</p> <p>OV = list returned completely filled, but more system variables than could fit in the list exist. Typically used to indicate "more" system variables in page at a time style list displays</p> <p>NR = list was returned empty. Last system variable in the list is returned as blanks.</p>	2	2		

Example

A user wants to print a list of all system variables.

```
DEF_LIST NAME(#VARLST) FIELDS(#VARNAM #VARDES #VARPGI
TYPE(*WORKING) ENTRYS(1000)
```

```
***** -Define the report layout-
```

```

DEF_REPORT  PRT_FILE(QSYSPRT)
DEF_HEAD    NAME(#HEAD01) FIELDS(#TEXT #PAGE . . . )
DEF_LINE    NAME(#VARPRT) FIELDS(#VARNAM #VARDES #VARPG
*****      -Set start system variable to blanks-
CHANGE      FIELD(#STRVAR) TO(*BLANKS)
*****      -Get list of system variables-
USE          BUILTIN(GET_SYSVAR_LIST) WITH_ARGS(#STRVAR) TO_C
*****      -If return code is OK then process list-
IF          COND('#RETCOD *EQ OK')
SELECTLIST  NAMED(#VARLST)
*****      -Print system variables-
PRINT       LINE(#VARPRT)
ENDSELECT
*****      -Otherwise issue an error-
ELSE
MESSAGE     MSGTXT('An error has occurred. Report not produced.')
ENDIF
*****      -Close printer file-
ENDPRINT

```

9.133 GET_TASK_DETAILS

⇒ **Note:** Built-In Function Rules..

Retrieves a list of all the objects that have been worked on or modified under the specified task.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Task Identifier NB. Value must be right justified.	1	8		
2	A	Opt	Object Type	2	2		
3	A	Opt	Object Name	1	10		
4	A	Opt	Object Extension NB. Object Type, Name, Extension would typically only be used to continue loading the list after a previous return code of OV.	1	10		

Return Values

--	--	--	--	--	--	--	--

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing objects modified by the input task. The list must contain no more than 32,767 entries and with details in the following format: From - To Description 1 - 2 Object Type 3 - 12 Object Name 13 - 22 Object Extension 23 - 72 Object Description 73 - 87 PC Name	22	87		
2	A	Req	The last Object Type in the returned list.	2	2		
3	A	Req	The last Object Name in the returned list.	1	10		
4	A	Req	The last Object Extension in the returned list.	1	10		
5	A	Req	Return Code OK = list returned partially or completely filled with task details. No more Task Details exist beyond those returned in the list. OV = list returned completely filled, but more Task Details than could fit in the list still exist. NR = list was returned empty. Last Object Type, Name and Extension in list returned as blanks. ER = ended in error.	2	2		

9.134 GET_TASK_LIST

⇒ **Note:** Built-In Function Rules.

Reads one of the LANSAs internal tables.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Task Identifier NB. Value must be right justified.	1	8		
2	A	Opt	Task Status Values: OPN = Open (No Work done yet) WRK = Open (Work has been done) CLS = Closed (Work completed Objects still locked to the task) FIN = Finished (Objects no longer locked) If no value specified all tasks are returned. This is the default.	3	3		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing tasks. The list must contain no more than 32,767 entries and with entries in the following format: From - To Description 1 - 8 Task Identifier 9 - 58 Task Description	58	58		
2	A	Req	The last Task Identifier in the returned list.	1	8		
3	A	Req	Return Code OK = list returned partially or completely filled with tasks. No more Tasks exist beyond those returned in the list. OV = list returned completely filled, but more Tasks than could fit in the list still exist. NR = list was returned empty. Last Task Identifier in list returned as blanks.	2	2		

9.135 GET_TEMPLATE_LIST

⇒ **Note:** Built-In Function Rules.

Returns a list of all templates in the system.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Search Template	1	10		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list containing Template details. The list must not contain more than 32,676 entries and entries must be in the following format: From - To Description 1 - 10 Template Name 11 - 50 Template Description	50	50		
2	A	Req	The last Template in the returned list.	1	10		

3	A	Req	<p>Return Code</p> <p>OK = list returned partially or completely filled with template details. No more Templates exist beyond those returned in the list.</p> <p>OV = list returned completely filled, but more Templates than could fit in the list still exist.</p> <p>NR = list was returned empty. Last Template in list returned as blanks.</p>	2	2		
---	---	-----	--	---	---	--	--

9.136 GET_WEB_COMPONENT

⇒ **Note:** Built-In Function Rules.

Get the page text (version 0) for a Web Component. Refer to the LANSAs for the Web Guide for more information about Web Components.

Use this Built-In Function with [9.198 PUT_WEB_COMPONENT](#) when you want to generate handcrafted web components at execution time.

For use with

LANSAs for i	YES	Only available for RDMLX.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Web Component	1	20		
2	A	Req	Mode Possible values are: I: Input O: Output N: Not applicable	1	1		
3	A	Opt	Language Default is current language For non-multilingual partitions this should be 'NAT' *DFT - partition default language.	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Possible values are: OK: Get completed normally. OV: list returned completely filled, but more lines of page text than could fit in the list still exist. NR: List is empty as component is not of a type that is supported by PUT_WEB_COMPONENT ER: Get encountered an error.	1	2		
2	L	Req	Working list to contain component source.	1	255		
3	A	Req	Page Generally the same as a Web component.	1	20		
4	A	Opt	Type. Possible values are: P: Page S: Script T: Text V: Visual If the component is one of the next three types, the list will be returned empty. W: Web Link B: Banner F: File	1	1		
5	A	Opt	Description	1	40		

9.137 HEXTOBIN

⇒ **Note:** Built-In Function Rules.

Converts the alphanumeric source string to its binary format. Each pair of characters in the source will be converted into its binary equivalent.

The source string should contain only characters 0-9, A-F. Any invalid characters will be treated as zero and an error status returned. The source string must be a multiple of two in length.

For example, source contains alphanumeric string C1C2 (IBM i) 4142 (Windows), Return is AB

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Source: Must contain only characters 0-9, A-F Must be a multiple of 2 in length.	2	Unlimited		
2	A	Opt	Y = Return true binary value. N = (Default) Return "string like" binary value. The first NULL byte in the return will be the terminator of the string.	1	1		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	w	Req	Returned binary value	1	Unlimited		
2	A	Opt	Return code. OK = action completed. ER = An error occurred.	2	2		

Technical Note

If 'Y' is specified for the second argument, use a Binary (or Alpha) field to get the returned value. Any other field type may cause the returned value to be truncated or unusable.

Example:

```
FUNCTION OPTIONS(*DIRECT)
DEFINE FIELD(#MYHEX) TYPE(*CHAR) LENGTH(100)
DEFINE FIELD(#MYHEX2) TYPE(*CHAR) LENGTH(100)
DEFINE FIELD(#MYRET) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#MYLEN) TYPE(*DEC) LENGTH(3) DECIMALS(0)
```

```
CHANGE FIELD(#MYHEX)
TO("414D5120414D5359443337202020002044B826A420C12563")
```

```
USE BUILTIN(HEXTOBIN) WITH_ARGS(#MYHEX 'Y')
TO_GET(#MYBIN #MYRET) (1)
CHANGE FIELD(#MYLEN) TO('24') (3)
USE BUILTIN(BINTOHEX) WITH_ARGS(#MYBIN #MYLEN)
TO_GET(#MYHEX2 #MYRET) (2)
RETURN
```

If #MYBIN is a Binary (or Alpha) field and is equal to or greater than 24 bytes, the BINTOHEX in (2) will return #MYHEX2 with same value as #MYHEX has. The value in #MYRET is 'OK'

If #MYBIN is a String, the BINTOHEX in (2) will return nothing to

#MYHEX2 and the value in #MYRET will be 'ER'.

Reason: Although the HEXTOBIN in (1) is instructed to return true binary value, the String nature of #MYBIN in this case will prevent it holding the full result. In fact #MYBIN will only get the binary form of '414D5120414D5359443337202020' as the next '00' is considered as a string terminator. So for the BINTOHEX in (2) #MYBIN has only 14 bytes, which is shorter than the 24 bytes expected.

9.138 IMPORT_OBJECTS

Acts as an interface to the LANSAs Import Facility.

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	YES

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Path that contains file(s) for IMPORT	1	256		
2	A	Opt	Run LANSAs Import with standard prompts Values: Y - LANSAs Import run with prompts N - LANSAs Import run without prompts Default = N Note: N implies that the LANSAs Import window will not require a user response on completion. The message to continue with the LANSAs Import after conversion to the intermediate format will not be displayed.	1	1		
3	A	Opt	Apply Filter to Import This is an array of single byte entries that controls the import of the associated Object Group. Byte: 1 = System Definition 2 = Partition Definition 3 = System Objects	1	256		

			<p>4 = User / Development See below for definition of objects Values : Y = Import this object group N = Force Import to bypass this object group Default = Y (For each entry if the argument is passed.)</p>				
4	A	Opt	<p>File Library Determines the value of the importing file's library to be saved into the repository. By specifying a value of I the importing file's library will be saved with the same file's library value as the exporting system. Setting this option to P will override the importing Files' File Library with the destination's Partition Data Library. Any File Library Substitutions setting will not be applied. Values: I – Use importing file's library P – Partition data library O – Use file's library override if included in the export information Default = I</p>	1	1		
5	A	Opt	<p>Allow Name Changes This parameter determines if the existing objects' names will be overwritten by the importing names. Note – Use this option with care as it will change existing objects and may impact other objects which refer to this name such as fields and files. The name may also be directly referenced in RDMLX code. Values: Y – Allow to override existing names</p>	1	1		

			N – Do not allow to override existing names. Default = N				
6	A	Opt	<p>Allow Type Change</p> <p>This parameter determines if the existing objects will be deleted and imported as a different object type. Note: Use this option with care as it will change existing objects and it may impact other objects which refer to this name such as fields and files. The name may also be directly referenced in RDMLX code.</p> <p>Values: Y - Delete existing object(s) and continue with import N - Do not proceed further if the importing Identifier is already in use by a different object type</p> <p>Default = N</p>	1	1		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1		Req	<p>Return Code</p> <p>OK = The import completed without error</p> <p>ER = A fatal error occurred during the import.</p>	2	2		
2	N	Opt	Number of Errors	1	7	0	0
3	N	Opt	Number of Warnings	1	7	0	0

Object Groups

System Definition

- System Options and Defaults
- LANSAs Commands

Partition Definition

- Partition Definition
- Partition Languages
- Groups and Frameworks

System Objects

- Users
- Enrolled Workstations

User / Development Objects

- Fields
- Files
- Processes
- Functions
- Components
- WAMs
- Weblets
- Technology Service Providers
- Technology Services
- Web Components
- Multilingual Variables
- System Variables
- Tasks

9.139 INSERT_IN_SPACE

⇒ **Note:** Built-In Function Rules.

Inserts a set of cell values into a space object. Refer also to the other [SPACE](#) Built-In Functions.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name	1	256		
2-20	w	O	Fields whose values are to be inserted into the space cells. The fields used do not have to be the same as the field names used to prototype the cell. It is the values of the fields specified in these arguments that are mapped into the space cells.	1	256	0	9

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "OK" = Cell values inserted successfully.	2	2		

		"ER" = Insert attempt failed. Messages issued will indicate more about the cause of the failure.				
--	--	--	--	--	--	--

Technical Notes

The field values must be specified in the same order as the cells in the space were defined. Cells are matched by the order of their specification in arguments 2 -> 20. The names of the fields used have no bearing whatsoever on the cell mapping logic.

If you specify less field values than there are cells in the space then the non-specified cells are set to blank/zero/null values as appropriate.

If you specify more field values than there are cells in the space then the additional field values are ignored.

The provided field values **MUST** be enough to uniquely identify the entry being inserted.

Examples

This example inserts the current values of the fields #EMPNO, #GIVENAME and #SURNAME into a space whose name is contained in field #SPACENAME:

```
Use Insert_in_Space (#SpaceName #Empno #GiveName #SurName)
```

This example inserts literal values into a space named TEST:

```
Use Insert_in_Space (TEST A0090 'Mary' 'Jones');
```

9.140 ISSUEINQUIRY

⇒ **Note:** Built-In Function Rules.

Issues an inquiry message to a message queue.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	ISSUEINQUIRY will only allow an alphanumeric reply value up to 20 characters in length
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Message queue to which inquiry is to be sent	1	10		
2	A	Req	Type of message (T = text, M = from message file)	1	1		
3	A	Req	Text of message (T) or message number (M)	1	132		
4	A	Opt	Message file name (M only) Default: QUSRMSG	1	10		
5	A	Opt	Message file library (M only) Default: *LIBL	1	10		
6	A	Opt	Message data Default: *BLANKS	1	132		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Reply to message input by message receiver	1	132		

Examples

Issue a message to ask who is signed on to a terminal:

```
USE BUILTIN(ISSUEINQUIRY) WITH_ARGS(DSP16 T 'Who are you ?')
  TO_GET(#REPLY)
```

Issue a message to ask operator if MYJOB can be submitted at this time:

```
CHANGE FIELD(#MSGID) TO(MSG0120)
CHANGE FIELD(#MSGF) TO(OPRMSGF)
CHANGE FIELD(#SUB) TO('MYJOB')
```

```
USE BUILTIN(ISSUEINQUIRY) WITH_ARGS(DSP01 M #MSGID #MSG
  '*LIBL' #SUB) TO_GET(#REPLY)
IF COND(#REPLY *EQ Y)
SUBMIT PROCESS(BTCHJOB) FUNCTION(UPDATE)
ELSE
MESSAGE MSGTXT('Job not submitted')
ENDIF
```

MSG0120 = Can &1 be submitted at this time ?

Note: The message data parameter corresponds directly to the MSGDTA parameter in the CL command SNDUSRMSG. The passing of information through this parameter is your responsibility. For more information on passing substitution variables see the appropriate IBM manuals.

9.141 ISSUEMESSAGE

⇒ **Note:** Built-In Function Rules.

Issues a message to a message queue.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Message queue to which message is to be sent. Do not leave this argument blank. Note: This argument is ignored on non-IBM i platforms.	1	10		
2	A	Req	Type of message (T = text, M = from message file)	1	1		
3	A	Req	Text of message (T) or message number (M)	1	132		
4	A	Opt	Message file name (M only) Default: QUSRMSG	1	10		
5	A	Opt	Message file library (M only) Default: *LIBL	1	10		
6	A	Opt	Message data Default: *BLANKS	1	132		

Return Values

No return values.

Examples

Issue a message to ask user to sign off for 10 minutes:

```
USE    BUILTIN(ISSUEMESSAGE)
      WITH_ARGS(DSP16 T
      'Please sign off for 10 minutes. Thank you.'
```

Issue a message to tell operator that DAYJOB to CLOSE has been submitted to the batch queue:

```
CHANGE  FIELD(#MSGID) TO(MSG0121)
CHANGE  FIELD(#MSGF) TO(WRKMSGF)
CHANGE  FIELD(#SUB) TO("DAILY  CLOSE")
*
SUBMIT  PROCESS(DAYLY) FUNCTION(CLOSE)
USE     BUILTIN(ISSUEMESSAGE) WITH_ARGS(DSP01 M #MSGID
      #MSGF #SUB)
*
MSG0121 = The &1 &2 job for today has been submitted.
```

Note: The message data parameter corresponds directly to the MSGDTA parameter in the CL command SNDPGMMSG. The passing of information through this parameter is your responsibility. For more information on passing substitution variables see the appropriate IBM manuals.

9.142 JSM_CLOSE

⇒ **Note:** Built-In Function Rules.

Closes the currently open connection to the JSM server.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	YES	Not available for RDMLX.
Visual LANSA for Linux	NO	Not available for RDMLX.

Arguments

None

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Status	1	20		
2	A	Req	Message	1	255		

Technical Notes

- JSM_CLOSE can only be used in an RDML function or component.

Example

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

9.143 JSM_COMMAND

⇒ **Note:** Built-In Function Rules.

Sends a command string to the currently open JSM server connection.

If an optional working list argument is supplied then the function fields and this working list contents are available to the loaded service.

Function field names prefixed with the letters JSM will not be sent, so it is recommended to use BIF argument names such as #JSMCMD, #JSMSTS and #JSMMSG.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	YES	Not available for RDMLX.
Visual LANSA for Linux	NO	Not available for RDMLX.

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Command	1	255		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Status	1	20		
2	A	Req	Message	1	255		
3	L	Opt	Working List This list must not have more than 9999 entries.	0	0		

Example

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

9.144 JSM_OPEN

⇒ **Note:** Built-In Function Rules.

Opens a connection to the JSM server, which starts a service thread to handle commands to be sent by the JSM_COMMAND BIF.

If no server argument is supplied, then the JSM server host is obtained by reading the JSMCLDTA data area.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	YES	Not available for RDMLX.
Visual LANSA for Linux	NO	Not available for RDMLX.

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Server	1	50		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Status	1	20		
2	A	Req	Message	1	255		

Example

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
USE BUILTIN(JSM_OPEN) WITH_ARGS(#JSMSRV) TO_GET(#JSMSTS #
```


9.145 JSMX_CLOSE

⇒ **Note:** Built-In Function Rules.

This Built-In Function closes the JSM connection identified by the handle.

See also: [9.147 JSMX_OPEN](#) and [9.146 JSMX_COMMAND](#)

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	Only available for RDMLX.
Visual LANSA for Linux	YES	Only available for RDMLX.

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Handle to connection	4	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	Status	1	20		
2	U	Req	Message	1	Unlimited		

Example

Close a JSM connection.

```
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMHNDL)  
TO_GET(#JSMSTS #JSMMSG)
```


9.146 JSMX_COMMAND

⇒ **Note:** Built-In Function Rules.

This Built-In Function sends a command string to the JSM connection identified by the handle.

If an optional working list argument is specified then the fields defined in that list are available to the loaded service. If no working list argument is specified then no fields are available to the loaded service. Note that this working list does not need to have any entries.

If an optional working list return value is specified then that working list is available to the loaded service.

See also: [9.147 JSMX_OPEN](#) and [9.145 JSMX_CLOSE](#).

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	Only available for RDMLX.
Visual LANSA for Linux	YES	Only available for RDMLX.

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Handle to connection	4	4		
2	U	Req	Command	1	Unlimited		
3	L	Opt	List of field definitions to send/receive. If this list is not passed, no fields will be used.				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	Status	1	20		
2	U	Req	Message	1	Unlimited		
3	L	Opt	Working List				

Technical Note 1

- There is a new keyword: SERVICE_CONTENT
- It can take one attribute: *HTTP
- When JSMX_COMMAND sends a SERVICE_LOAD command it may also use the SERVICE_CONTENT keyword.

Note that the following will ONLY happen if you are running under the context of JSMDirect

If SERVICE_CONTENT(*HTTP) is used in the command string, the following will happen:

If this is the first connection to use SERVICE_CONTENT(*HTTP)
read STDIN (this can only be done once)

this connection takes ownership of JSM_CLOSE (receives STDOUT)

If this is NOT the first connection to use SERVICE_CONTENT(*HTTP)

this connection takes ownership of JSMX_CLOSE (receives STDOUT) - The previous connection loses the ownership.

Scenario A:

#1 JSMX_OPEN - open connection

#2 JSMX_OPEN - open connection

#3 JSMX_OPEN - open connection

#1 JSMX_COMMAND("SERVICE_LOAD(xxx)

SERVICE_CONTENT(*HTTP)") - send CGI keywords, read STDIN, claim ownership

#2 JSMX_COMMAND("SERVICE_LOAD(xxx)") - send CGI keywords

#3 JSMX_COMMAND("SERVICE_LOAD(xxx)

SERVICE_CONTENT(*HTTP)") - send CGI keywords, transfer ownership

#1 JSMX_CLOSE - close connection

#2 JSMX_CLOSE - close connection
#3 JSMX_CLOSE - close connection and write STDOUT

Scenario B:

#1 JSMX_OPEN - open connection
#1 JSMX_COMMAND("SERVICE_LOAD(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, read STDIN, claim
ownership
#1 JSMX_CLOSE - close connection and write STDOUT
#2 JSMX_OPEN - open connection
#2 JSMX_COMMAND("SERVICE_LOAD(xxx)") - send CGI keywords
#2 JSMX_CLOSE - close connection
#3 JSMX_OPEN - open connection
#3 JSMX_COMMAND("SERVICE_LOAD(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, transfer ownership
#3 JSMX_CLOSE - close connection

Note

Only one connection can have ownership of JSMX_CLOSE at a time.

STDIN is read once only.

STDOUT is written once only.

The CGI keywords are always sent.

Technical Note 2

The following datatypes are supported. If any other datatype is used, that datatype will be ignored, that is, not passed to JSM.

TYPE(*CHAR)

TYPE(*DATETIME)

TYPE(*BOOLEAN)

TYPE(*STRING)

TYPE(*INT)

TYPE(*TIME)

TYPE(*DATE)

TYPE(*PACKED)

TYPE(*DEC)

TYPE(*FLOAT)

TYPE (*SIGNED)

Example

Example: To call a command using field list and working list.

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMHNDL #JSMCMD  
#FLDLST) TO_GET(#JSMSTS #JSMMSG #WRKLST)
```

9.147 JSMX_OPEN

⇒ **Note:** Built-In Function Rules.

This Built-In Function opens a connection to the JSM server, which starts a service thread to handle commands to be sent by the JSMX_COMMAND BIF. JSMX_OPEN returns a handle, which identifies the JSM connection.

If no server argument is supplied, then the JSM server host is obtained by reading the JSMCLDTA data area.

See also: [9.146 JSMX_COMMAND](#) and [9.145 JSMX_CLOSE](#).

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	Only available for RDMLX.
Visual LANSA for Linux	YES	Only available for RDMLX.

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Server (host:port), Default is *BLANKS	1	50		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	Status	1	20		
2	U	Req	Message	1	Unlimited		
3	A	Req	Handle to connection	4	4		

Examples

Example 1: Open a connection using server in JSMCLTDTA data area.

```
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMSTS #JSMMSG #JSMHDNL)
```

Example 2: Open a connection specifying a server.

```
USE BUILTIN(JSMX_OPEN) WITH_ARGS(#JSMSRV)  
TO_GET(#JSMSTS #JSMMSG #JSMHDNL)
```

9.148 LEFT

⇒ **Note:** Built-In Function Rules.

Left aligns argument into return string.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be left align	1	256		
2	A	Opt	Remove imbedded blanks flag (Y/N) Values: Y = remove N = do not remove Default: N	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return left align string	1	256		

Example

Left align and remove imbedded blanks from a requested string.

```
DEFINE FIELD(#INTEXT) TYPE(*CHAR) LENGTH(18)
DEFINE FIELD(#OUTTEXT) TYPE(*CHAR) LENGTH(18)
*****
REQUEST FIELDS(#INTEXT)
USE BUILTIN(LEFT) WITH_ARGS(#INTEXT Y) TO_GET(#OUTTEXT)
DISPLAY FIELDS(#OUTTEXT)
```

Resulting displays would look something like the following.

```
      FUN01      Left Example
*
In text . . .  FR  E  D
*
CF1=Help
*
```

then,

```
      FUN01      Left Example
*
Out text . . . FRED
*
CF1=Help
*
```

9.149 LIST_PRINTERS

⇒ **Note:** Built-In Function Rules.

This BIF will return a list of printers currently configured on the machine.

Refer to Technical Notes if running on Windows Vista.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Printer Location A – All printers L – Local Only - use always with IBM i. Default is A.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working List containing the printer's full name (see technical notes). The list must be in the following format: From – To Description	255	255		

			1 – 255 Printer Name 256 – 256 Printer Location (L – local, N – network).				
2	A	Req	Return Code OK = list returned successfully OV = list returned completely filled but more printers than could fit in the list still exist. NR = empty list returned ER = an error occurred	2	2		

Technical Notes

- The working list will return the full printer name. This includes the domain name for remote printers. That is //domain/PrinterName. For example, \\ourdomain\Epson Stylus COLOR 900.
- If printing through a server function, this BIF should be run on the server to obtain the list of printers available to the server and the list then passed back to the client. The selected printer name may then be sent to the server function that will perform the printing.
- If running on Windows Vista, there is a limit to the number of printers that can be defined using this Built-In Function. This includes both local and remote printers. The limit depends on the printers defined, but as a guide, you should limit the number of printers to 12.

Example

```
FUNCTION OPTIONS(*DIRECT)
DEF_LIST NAME(#PRNLIST) FIELDS(#PRN_NAME #PRN_LOC)
TYPE(*WORKING)
USE BUILTIN(LIST_PRINTERS) WITH_ARGS(A) TO_GET(#PRNLIST
#STD_CMPAR)
```

9.150 LOAD_FILE_DATA

⇒ **Note:** Built-In Function Rules.

Will call the OAM for the requested file and load all the data from the flat file specified.

This Built-In Function expects to be executed on the same machine as the OAM. Both the BIF and the OAM need to access the input file. If you execute the BIF from a local Function but redirect the File to SuperServer, it is your responsibility to ensure that the input filename is valid on both the client and the server.

Of course, while loading data into a file it is not expected that an IO is still in use from, say, calling LOAD_FILE_DATA from within a SELECT loop over the same file! At the completion of loading the file the OAM is completely closed, terminating all existing transactions.

For use with

LANSA for i	NO	This function is not supported on IBM i. If executed on IBM i, a fatal error will result with this message "This Built-In Function is not supported in the current release".
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No.	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
-----	------	---------	-------------	---------	---------	---------	---------

1	A	Req	The LANSA File Name	1	10		
2	A	Opt	Use Rules/Triggers ? Y will cause all rules, triggers and batch control logic to be processed, any other value will load the data without processing rules, triggers and batch control logic. Default = N	1	1		
3	A	Opt	Input File Path/Name Default Value =\X_ppp\source\[File Name].dat	1	256		
4	A	Opt	Ignore Duplicate Keys Y - duplicate keys found when attempting to add a record will cause the record to not be added and a message to be added to the job message queue. N - duplicate keys will cause the load to end with an error Default = Y				
5	A	Opt	Y\N Check for OAM Default = N	1	1		
6	A	Opt	Delete .dat file and BLOB/CLOB files after a successful load (no errors and no warnings about missing BLOB and CLOB data). Default - N.	1	1		
7	A	Opt	If BLOB or CLOB field exist on the file, and the .dat file indicates a filename for the field, LOAD_FILE_DATA will automatically look for the file and load the data into the database if the file is found. This flag controls what to do if an expected file is not found. Default = W, meaning issue a warning message and set field to default (*SQLNULL).	1	1		

			If Y, set field to default (*SQLNULL). If N, give a fatal error and abort.				
8	A	Opt	CTD Location Level A= All (Partition + System). P = Partition Level only. S=System Level only. Default is A.	1	1		

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code OK = File successfully loaded VE = Validation Error in load of the file ER = File load failed (possible causes - flat file not found, a trigger failed to execute correctly) NT = No table exists NO = No OAM. File does not exist or is not compiled. NO only returned when input option 5 is set to Y	2	2		

Issues related to OAMs built prior to V10.0

When UNLOAD_FILE_DATA has been used with an OAM built prior to V10.0, then a CTX file must exist for V10.0 LOAD_FILE_DATA to work. Note that the following steps to manually create a CTX file are not required when data is created by a V10.0 OAM.

Follow these steps to create a CTX file. V9.1 to V10.0 has been used in this example:

1. Return to V9.1 of LANSAs that generated the data file.
2. Make a copy of the relevant V9.1 .CTD (Common Table Definition) file with an extension of .CTX (ex-Common Table Definition).
3. Copy the V9.1 .CTX file into: ...\`x_win95`\X_lansa\X_<partition>\Source\
For example: ...\`x_win95`\X_lansa\X_<partition>\Source\`<filename>`.CTX
4. Use the LOAD_FILE_DATA Built-In Function to load the unloaded data.
5. Delete the CTX file after using LOAD_FILE_DATA as it is no longer needed.

9.151 LOAD_OTHER_FILE

⇒ **Note:** Built-In Function Rules.

Loads the definition of an "OTHER" file.

An edit session must be commenced by using the START_FILE_EDIT Built-In Function, prior to using this Built-In Function.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Maximum number of logical files to load Default: 5	1	2	0	0
2	L	Opt	Working list to contain the names of logical files to be loaded. Each logical file name must be 10 bytes. If this argument is provided, then Argument 1 (Maximum number of logical files to load) must be zero.	100	100		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec

1	A	Req	Return code: OK, ER. In case of "ER" return code, error message(s) are issued automatically.	2	2		
---	---	-----	---	---	---	--	--

Example

A user wants to control the load of an "OTHER" file definition using their own version of the 'Load an "OTHER" file' option. A maximum of 2 logicals to load has been specified in this example.

```

***** Define arguments and lists
DEFINE  FIELD(#FILNAM) TYPE(*CHAR) LENGTH(10)
DEFINE  FIELD(#LIBNAM) TYPE(*CHAR) LENGTH(10)
DEFINE  FIELD(#MAXLOG) TYPE(*DEC) LENGTH(2) DECIMALS(0)
DEFINE  FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
BEGIN_LOOP
***** Request File and library name and maximum number of
***** logicals to load
REQUEST  FIELDS(#FILNAM #LIBNAM #MAXLOG)
*****
USE      BUILTIN(START_FILE_EDIT) WITH_ARGS(#FILNAM #LIBNAM)
USE      BUILTIN(LOAD_OTHER_FILE) WITH_ARGS(2) TO_GET(#RETCOD)
USE      BUILTIN(END_FILE_EDIT) WITH_ARGS('Y') TO_GET(#RETCOD)
***** Submit job to make file operational
USE      BUILTIN(MAKE_FILE_OPERATIONL) WITH_ARGS(#FILNAM #LIBNAM)
*****
END_LOOP

```

9.152 LOCK_OBJECT

⇒ **Note:** Built-In Function Rules.

Attempts to place a lock on the specified User Object and returns an error if unsuccessful.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object Type.	1	20		
2	A	Opt	Object Identifier 1.	1	10		
3	A	Opt	Object Identifier 2.	1	10		
4	A	Opt	Object Identifier 3.	1	10		
5	A	Opt	Object Identifier 4.	1	10		
6	A	Opt	Locking Level FUNC = Function (Default) JOB = Job PERM = Permanent	3	4		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Object was successfully locked. IG = Lock request ignored. Object already locked by current Job. ER = Object already has a lock present.	2	2		
2	A	Opt	Job Name of Locking Job	1	10		
3	A	Opt	User of Locking Job	1	10		
4	A	Opt	Job Number of Locking Job	1	6		

This Built-In Function allows the locking of User Objects. Once a User Object is locked any other attempt to lock that User Object results in an error condition and a Return Code of "ER" being returned. The exception to this is when an attempt is made to lock a User Object that is already locked by the current job. When this occurs the lock request is ignored and a Return Code of "IG" is return.

A User Object need not be a physical object on the system, it can be ANYTHING you require. User Objects are more conceptual than physical, and can represent a particular customer number or an entire LANSAs partition. The only limitation is your imagination.

The word 'ALL' is reserved by the system and must not be used as an Object Identifier.

This method of locking is controlled by your application, it is NOT controlled by the operating system. Therefore system commands cannot be used to investigate the lock status of User Objects.

Some of the advantages of using LOCK_OBJECT over the LOCK(*YES) parameter of I/O commands include:

- You can devise a proper locking "protocol". For example LOCK_OBJECT ('ORDER' #ORDNUM is the "protocol" to lock the order identified by #ORDNUM. By using a protocol you do not have to worry/think about how your application will lock the order header, the associated line items and line

items charges across 3 differing database files.

- You do not introduce inter or intra operating system dependencies. For example IBM i will lock multiple records in the same file when full before/after image commitment control is used, but it won't in other situations.
- You can lock multiple objects easily.
- You can "invent" objects to handle any type of situation, you are no longer dependent on file "records".
- You can have "permanent" locks lasting days, months or years.

The Object Type argument is used to organize the object identifiers into groups of common attributes. For example an Object Type could be used to group customers, orders or printer names.

The Object Identifiers then specify a single object.

Current User Object locks are stored in the file DC@FOL. There exists one record in DC@FOL for each User Object lock present. The layout of DC@FOL is:

Field Name	Length	Description
FOLP#I	3A	Partition Identifier
FOLTYP	20A	Object Type
FOLID1	10A	Object Identifier 1
FOLID2	10A	Object Identifier 2
FOLID3	10A	Object Identifier 3
FOLID4	10A	Object Identifier 4
FOLLVL	4A	Lock Level
FOLMOD	10A	Process name
FOLFMT	7A	Function name
FOLJNL	10A	Locked by Job Name
FOLJ#L	6A	Locked by Job Number
FOLUSL	10A	Locked by User
FOLTDS	12S 0	Time/Date Stamp - System Format

The Logical Views present and their keys are:-

Logical File	Key Sequence	Use
DC@FOLV1	FOLP#I, FOLTYP, FOLID1, FOLID2, FOLID3, FOLID4.	Read Only
DC@FOLV2	Same as DC@FOLV1.	Update
DC@FOLV3	FOLLVL, FOLJ#L, FOLMOD, FOLFMT, FOLP#I, FOLTYP, FOLID1, FOLID2, FOLID3, FOLID4.	Read Only
DC@FOLV4	Same as DC@FOLV3.	Update

Be aware that backup and recovery of this file is a user responsibility. Any locks that are present when a backup is done will be reinstated when that backup is restored.

Should LANSAP or a function crash due to error any locks that are present will be removed. But if the system were to crash, LANSAP will be unable to remove the locks. Locks remaining due to a system crash become the responsibility of the user to remove.

The User Object locks are either unlocked automatically or when the UNLOCK_OBJECT Built-In Function is used.

The Locking Level used on the LOCK_OBJECT determines when the User Object would be automatically unlocked.

- A Locking Level of 'FUNC' indicates that the lock will be automatically removed at the end of the function that created it.
- A Locking Level of 'JOB' indicates that the lock will be automatically removed when you exit JOB, for example, a form or process.
- If you need a User Object lock to exist after you have exited LANSAP then a Locking Level of 'PERM' is required. This is a permanent User Object lock that exists until explicitly removed with UNLOCK_OBJECT.

Examples

By using the four User Object Identifiers to build a structure to your object locking it is possible to do generic key unlocking.

```
DEFINE FIELD(#RETURN) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#STATE) TYPE(*CHAR) LENGTH(3)
DEFINE FIELD(#CUSTNO) TYPE(*CHAR) LENGTH(6)
***** Lock Customers to be Updated
USE BUILTIN(LOCK_OBJECT)
    WITH_ARGS('CUSTOMER' #STATE #CUSTNO " " 'FUNC') TO_GET(#STATE)
***** Unlock ALL Customers for the STATE
USE BUILTIN(UNLOCK_OBJECT) WITH_ARGS('CUSTOMER' #STATE #CUSTNO
    TO_GET(#RETURN))
```

In this example a number of User Objects relating to customers had locks placed on them. All the locked 'customers' belong to the same 'state'. When the update process is complete the locks can be removed. Rather than remove each individual lock for each customer, all the locks for a state can be removed generically by specifying 'ALL' on the second Object Identifier.

User Object locking can also be used to limit the number of concurrent users a function can have. For example, if you wish to impose a three user limit on your order entry function, then create three User Objects, each relating to a user. Then at the top of your order entry function attempt to obtain a lock on one of the User Objects. If none are available then you could display a message stating this, and exit the function. If a lock is granted then allow access to the function. Remember to unlock the User Object when exiting the function, or specify 'FUNC' as the Locking Level so the lock will be automatically removed when the function is finished. Use the following example as a guide.

```
DEFINE FIELD(#RETURN) TYPE(*CHAR) LENGTH(2)
***** Attempt lock on 1st instance
USE BUILTIN(LOCK_OBJECT)
    WITH_ARGS('ORDER_ENTRY' 'ORDER#1' " " " 'FUNC') TO_GET(#STATE)
IF COND(#RETURN *EQ ER)
***** Attempt lock on 2nd instance
USE BUILTIN(LOCK_OBJECT)
    WITH_ARGS('ORDER_ENTRY' 'ORDER#2' " " " 'FUNC') TO_GET(#STATE)
IF COND(#RETURN *EQ ER)
***** Attempt lock on 3rd instance
USE BUILTIN(LOCK_OBJECT)
    WITH_ARGS('ORDER_ENTRY' 'ORDER#3' " " " 'FUNC') TO_GET(#STATE)
```

```
IF      COND('#RETURN *EQ ER')
***** Cannot obtain any locks
MESSAGE MSGTXT('No Order Entry sessions are Available')
RETURN
ENDIF
ENDIF
ENDIF

***** Protected processing
***** Unlock ALL Order Entry locks for this Function
USE     BUILTIN(UNLOCK_OBJECT) WITH_ARGS('ORDER_ENTRY' 'A
```

9.153 LOGICAL_KEY

⇒ **Note:** Built-In Function Rules.

Specifies or re-specifies the name of a field that is a key of a logical view / file previously defined by the LOGICAL_VIEW Built-In Function.

Prior to using this Built-In Function an edit session must be commenced by using the START_FILE_EDIT Built-In Function .

Allowable argument values and adopted default values are as described in [Detailed Logical View Maintenance](#) in the *LANSA for i User Guide*.

Warning: This Built-In Function cannot be used for a file of type "OTHER".

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of logical view to which key is to belong.	1	10		
2	A	Req	Name of key field. Must have been previously specified as a field in the file by using the FILE_FIELD Built-In Function.	1	10		
3	N	Opt	Optional sequencing number. Used to sequence key fields. If not specified keys are sequenced in the same order as they are presented.	1	5	0	0

4	A	Opt	Ascending or Descending key. Must be A or D. Default is A.	1	1		
5	A	Opt	Signed, Unsigned or Absolute value ordering of numeric key. Must be S,U or A. Default is U for alphas and S for numerics.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = key defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically and the edit session ended without commitment.	2	2		

9.154 LOGICAL_VIEW

⇒ **Note:** Built-In Function Rules.

Specifies or re-specifies the name and basic attributes of a logical view / file that is to base on the file definition being edited.

Prior to using this Built-In Function an edit session must be commenced by using the START_FILE_EDIT Built-In Function.

After using this Built-In Function to define the basic logical view / file attributes, repetitively use the LOGICAL_KEY Built-In Function to specify or re-specify the key field name(s).

Allowable argument values and adopted default values are as described in [Detailed Logical View Maintenance](#) in the *LANSA for i User Guide*.

Warning: This Built-In Function cannot be used for a file of type "OTHER".

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of logical view.	1	10		
2	A	Req	Description of logical view	1	40		
3	A	Opt	Access path maintenance option Must be IMMED or DELAY. Default is IMMED.	1	7		
4	A	Opt	Uniquely keyed file / view Must be YES or	1	3		

			NO. Default is NO.				
5	A	Opt	Dynamic record selection Must be Y or N. Default is N. Must be YES or NO. Default is NO.	1	3		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = logical view defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically and the edit session ended without commitment.	2	2		

9.155 MAIL_ADD_ATTACHMENT

⇒ **Note:** Built-In Function Rules.

This Email handling Built-In Function is used to add a file to be sent as an attachment to the current email.

Successive calls enable an internal list of attachments for sending to be built-up.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	Refer to Email Built-In Function Notes before using this Built-In Function.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Full path to file including directories (Windows) and file.	1	255		
2	A	Opt	The attachment filename seen by the recipient, which may differ from the filename in the Full Path argument if temporary files are being used. Default behavior is mail system specific.	1	255		
3	A	Opt	Reserved for future use. This argument is currently unused.	0	10		
4	N	Opt	Reserved for future use. This argument is currently unused.	5	5		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK - Action completed ER - Error occurred	2	2		

Technical Notes

- No attachments are defined immediately following a MAIL_START call.
- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- If any error occurs, all details of the email definition will be lost. To restart processing, a new call to MAIL_START would be required.

Example

The following example shows only this function. See the example for MAIL_START which defines all details of an Email message and then sends it by using Built-In Functions.

```
***** COMMENT(Define attachment file)  
USE BUILTIN(MAIL_ADD_ATTACHMENT) WITH_ARGS('c:\config.sys' ']
```

9.156 MAIL_ADD_ORIGINATOR

⇒ **Note:** Built-In Function Rules.

This Email Handling Built-In Function is used to add the name of the original sender for the current email.

This call will allow you to specify the original sender for the message.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	Refer to Email Built-In Function Notes before using this Built-In Function.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Display name of the message sender.	1	255		
2	A	Opt	Sender's address. This address is provider-specific message delivery data. For outbound messages, this argument may be an address entered by the user for a sender that is not in an address book (that is, a custom sender). Default behavior is mail system specific.	1	255		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
----	------	-------------	-------------	------------	------------	------------	------------

1	A	Opt	Return Code	2	2		
			OK - Action completed				
			ER - Error occurred				

Technical Notes

- On Windows systems, the mail system will normally use the current mail user as the original sender and will ignore this setting but no error will occur.
- On IBM i, a mail message requires a minimum of one originator (normally you would only provide one originator). If no originator is provided, the current user's email address is used. The current user must be registered in the SNADS directory and have a SMTP alias.
- Under IBM i, enter the full email address (the address type prefix SMTP: is optional) in either the display name argument or the sender's address argument if a display name is specified.
- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- If any error occurs, all details of the email definition will be lost. To restart processing, a new call to MAIL_START would be required.

Example

This example shows only this function. See the example for MAIL_START which defines all details of an Email message and then sends it by using Built-In Functions.

Note: If you wish to use this example for testing you will need to insert a valid recipient's address in the required argument.

```
***** COMMENT(Define Message Originator if IBM i)
***** COMMENT(MAPI on non-IBM i usually defaults)
IF COND(*HOST)
USE BUILTIN(MAIL_ADD_ORIGINATOR) WITH_ARGS('SMTP:<-
- recipient@address -->') TO_GET(#LEM_RETC)
ENDIF
```

9.157 MAIL_ADD_RECIPIENT

⇒ **Note:** Built-In Function Rules.

This Email handling Built-In Function is used to add the name of a recipient for the current email.

Successive calls enable an internal list of recipients to be built up. As a minimum you would normally define at least one recipient of class "TO".

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	Refer to Email Built-In Function Notes before using this Built-In Function.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Recipient Class. Values: TO - To CC - Copy to BCC - Blind copy to	2	3		
2	A	Req	Display name of the message recipient.	1	255		
3	A	Opt	Recipient's address. This address is provider-specific message delivery data.	1	255		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK - Action completed ER - Error occurred	2	2		

Technical Notes

- No recipients are defined immediately following a MAIL_START call.
- The optional Recipient's address may be an address entered by the user for a recipient not in an address book (that is, a custom recipient). Default behavior is mail system specific. MAPI will typically require that you provide this address in the form:
<address type>:<full address>

For example:

Microsoft PC Mail: MS:network/postoffice/mailbox

Internet: SMTP:mailbox@companyname.com

- Under IBM i, enter the full email address (the address type prefix SMTP: is optional) in the display name argument or the sender's address argument if a display name is specified. A mail message requires a minimum of one recipient.
- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- If any error occurs, all details of the email definition will be destroyed. To restart processing, a new call to MAIL_START is required.

Example

This example shows only this function. Refer to the [9.162 MAIL_START](#) example, which defines all details of an Email message.

Note: If you wish to use this example for testing you should insert a valid display name as the second argument and insert a valid recipient's address as the third argument.

```
***** COMMENT(Set Recipient using TO argument)
***** COMMENT(may also set others for TO CC BCC)
lansaELSE
USE BUILTIN(MAIL_ADD_RECIPIENT) WITH_ARGS(TO '<-- name --
>' 'SMTP:<-- recipient@address -->')
TO_GET(#LEM_RETC)
```

9.158 MAIL_ADD_TEXT

⇒ **Note:** Built-In Function Rules.

This Email handling Built-In Function is used to add text to the message buffer for the current email.

Successive calls enable a message text to be built up by concatenating the text to the end of the buffer. Each call may determine if a newline is to be added to the end of the text in that call.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	Refer to the Email Built-In Function Notes before using this Built-In Function.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Text to be added to the email message	1	Unlimited		
2	A	Opt	Add new line to the end of the text. Values: Y - Yes N - No Default Y	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK - Action completed ER - Error occurred	2	2		

Technical Notes

- The message buffer is cleared when MAIL_START is called.
- Trailing blanks in the text are truncated. If you require blanks to be inserted between successive calls that have no intervening new line, place the blanks at the beginning of the later call.
- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- If any error occurs, all details of the email definition are destroyed. To restart processing, a new call to MAIL_START would be required.

For the maximum possible length of a field type please refer to [Field Type Considerations](#).

Example

This example shows only this function. See the example for MAIL_START which defines all details of an Email message and then sends it by using Built-In Functions.

```
***** COMMENT(Set message Text)
USE BUILTIN(MAIL_ADD_TEXT) WITH_ARGS('Hello,' Y) TO_GET(#LE
DEFINE FIELD(#BIGLINE) TYPE(*CHAR) LENGTH(255) LABEL('Big Te
CHANGE FIELD(#BIGLINE) TO("I am sending this message just to try out t
In Functions.")
USE BUILTIN(MAIL_ADD_TEXT) WITH_ARGS(#BIGLINE Y) TO_GET(
USE BUILTIN(MAIL_ADD_TEXT) WITH_ARGS('Thank You' N) TO_GET
```

9.159 MAIL_SEND

⇒ **Note:** Built-In Function Rules.

Used to send the constructed email message.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

None

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK - Action completed ER - Error occurred	2	2		

Technical Notes

- On the Visual LANSA platform, MAIL_SET_OPTION calls may be required to logon to the MAPI mail provider. The information set in this way is not used until a MAIL_SEND call is issued at which time an attempt at logon validation will take place.
- On IBM i the mail message is written to the IBM i Mail Server Framework (MSF). The mail message is then processed by a MSF job and delivered according to the schedule of the distribution queues. MSF must be configured

and running before any mail messages can be delivered.

- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- MAIL_SEND destroys the current mail information on return from the call even if the call is unsuccessful. To start processing a new message, a new call to MAIL_START is required.

Example

This example shows only this function. See the example for MAIL_START which defines all details of an Email message and then sends it by using Built-In Functions.

```
***** COMMENT(SEND the mail)
USE BUILTIN(MAIL_SEND) TO_GET(#LEM_RETC)
```

9.160 MAIL_SET_OPTION

⇒ **Note:** Built-In Function Rules.

Used to set various options which may be required by the mail system.
Refer to [Email Built-In Functions Notes](#) before using this Built-In Function.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

LANSA for i Specific Options

Option Name	Option Values	Option Default
PRIORITY An indication of the priority of the message.	'NORMAL' 'LOW' 'URGENT'	'NORMAL'
SENSITIVITY An indication of the sensitivity of the message content.	'NONE' 'PERSONAL' 'PRIVATE' 'CONFIDENTIAL'	'NONE'
IMPORTANCE An indication of the importance of the message content.	'NORMAL' 'LOW' 'HIGH'	'NORMAL'
MESSAGE_CC SID The CCSID to use for the mail message text strings such as mail message subject and text lines.	String	'65535'
CONTENT_TYPE The content-type that the message text contains.	String	'text/plain'

Examples are 'text/plain' & 'text/html'. Other values may be used, but in any case the application should be tested to ensure that the content-type is suitable. This option value is not validated, nor is the message text validated against the content-type. When using the content-type of 'text/html', be aware that some email clients may not be able to render this, also this may trip some spam filters.		
---	--	--

Visual LANSA Specific Options

Option Name	Option Values	Option Default
PROFILENAME Profile name string to use when logging on. If the value supplied is invalid, and MAPI_LOGON_UI is set, MAPI displays an error followed by a logon dialog box with an empty name field.	String	"Windows Messaging Settings"
PASSWORD Credential string. If the messaging system does not require password credentials, or if it requires that the user enter them, this option should NOT be set. When the user must enter credentials, the MAPI_LOGON_UI or MAPI_PASSWORD_UI option must be set to allow a logon dialog box to be displayed.	String	None - empty string
MAPI_NEW_SESSION Indicates if an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION option is not set, an existing shared session is used.	Y/N	N
MAPI_LOGON_UI Indicates if a logon dialog box should be displayed to prompt the user for logon information. If the user needs to provide a password and profile name to enable a successful	Y/N	N

logon, MAPI_LOGON_UI must be set.		
MAPI_PASSWORD_UI Indicates if MAPI should only prompt for a password and not allow the user to change the profile name. You should not set both MAPI_PASSWORD_UI and MAPI_LOGON_UI since the intent is to select between two different dialog boxes for logon.	Y/N	N

Options Applicable to All Platforms

These options may be used on any platform.

Option Name:	Option Values	Option Default
RECEIPT_REQUESTED Indicates if a receipt notification is requested.	Y/N	N

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Option Name (see descriptions above).	1	20		
2	A	Req	Option Value (see descriptions above).	1	255		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code	2	2		

		OK - Action completed				
		ER - Error occurred				

Technical Notes

- The MAPI interface is used on the Windows platform. MAIL_SET_OPTION calls may be required for logon to the MAPI mail provider which begins a session with the messaging system. To request the display of a logon dialog box if the credentials presented (password) fail to validate the session, set the MAPI_LOGON_UI or MAPI_PASSWORD_UI option. If you do not allow the display of the dialog box, all logon information, default or specified must be valid or the MAIL_SEND will fail during the MAPI logon phase with very little information available as to the cause of the failure. The information set in this way is not used until a MAIL_SEND call is issued at which time an attempt at logon validation will take place.
- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- If any error occurs, all details of the email definition will be lost. To restart processing, a new call to MAIL_START would be required.
- The PRIORITY, SENSITIVITY and IMPORTANCE mail options are usually ignored by PC mail clients. It is your responsibility to verify they work properly in your environment.
- The RECEIPT_REQUESTED: Y mail option may either be ignored or disabled by PC mail clients, or disabled at the recipient mail server to prevent spam. It is your responsibility to verify it works properly in your environment.

Example

This example shows only this function. See the example for MAIL_START which defines all details of an Email message and then sends it by using Built-In Functions.

```
***** COMMENT(Set receipt acknowledgement is required)
USE BUILTIN(MAIL_SET_OPTION) WITH_ARGS(RECEIPT_REQUESTE
```

9.161 MAIL_SET_SUBJECT

⇒ **Note:** Built-In Function Rules.

Used to set the text of the messages subject.

You would normally call this function to set a Subject but it is not mandatory.

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	Refer to Email Built-In Function Notes before using this Built-In Function.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Text string describing the message subject.	1	255		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK - Action completed ER - Error occurred	2	2		

Technical Notes

- No Subject is defined immediately following a MAIL_START call.
- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- If any error occurs, all details of the email definition will be lost. To restart processing, a new call to MAIL_START would be required.

Example

This example shows only this function. See the example for MAIL_START which defines all details of an Email message and then sends it by using Built-In Functions.

```
***** COMMENT(Set Subject text)
USE BUILTIN(MAIL_SET_SUBJECT) WITH_ARGS('Testing Email Built-
In Functions') TO_GET(#LEM_RETC)
```

9.162 MAIL_START

⇒ **Note:** Built-In Function Rules.

Used to start an email session.

The email session started can be used by the calling function to define and send a single email message.

Refer to [Email Built-In Functions Notes](#) before using this Built-In Function.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

None

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK - Action completed ER - Error occurred	2	2		

Technical Notes

- MAIL_START destroys any existing mail message information and starts a new mail session.
- Only one email definition can be in progress at one time (ie: it is not possible

to concurrently define two email definitions prior to sending them from within the same job).

- Details of the email definition will be lost unless the MAIL_SEND Built-In Function is used to send the message.
- If any error occurs, all details of the email definition are destroyed. To restart processing, a new call to MAIL_START would be required.

Example

This example defines all details of an Email message and then sends it by using Built-In Functions.

Note: If you wish to use this example for testing you should replace <-- name --> and <-- recipient@address --> where these are used in the examples of MAIL_ADD_RECIPIENT and MAIL_ADD_ORIGINATOR.

```
FUNCTION OPTIONS(*DIRECT)
DEFINE FIELD(#LEM_RETC) TYPE(*CHAR) LENGTH(2) LABEL('Return
DEF_COND NAME(*OKAY) COND('#LEM_RETC = OK')
DEF_COND NAME(*NOTOKAY) COND('#LEM_RETC *NE OK')
DEF_COND NAME(*CLIENT) COND('*CPUTYPE *NE AS400')
DEF_COND NAME(*HOST) COND('*CPUTYPE *EQ AS400')
*****
BEGIN_LOOP
***** COMMENT(Start Mail message - Initialize)
USE BUILTIN(MAIL_START) TO_GET(#LEM_RETC)
***** COMMENT(If not IBM i, set MAPI arguments)
IF COND(*CLIENT)
***** COMMENT(Use the Windows Default Profile)
USE BUILTIN(MAIL_SET_OPTION) WITH_ARGS('PROFILENAME' 'Win
***** COMMENT(Assume user has no Password)
USE BUILTIN(MAIL_SET_OPTION) WITH_ARGS('PASSWORD' *BLANK)
***** COMMENT(Use existing MAPI mail session)
***** COMMENT(from logged on Email client)
USE BUILTIN(MAIL_SET_OPTION) WITH_ARGS('MAPI_NEW_SESSION')
***** COMMENT(Display MAPI Logon dialog if required)
USE BUILTIN(MAIL_SET_OPTION) WITH_ARGS('MAPI_LOGON_UI' 'Y')
***** COMMENT(Do not display MAPI Password Dialog)
***** COMMENT(mutually exclusive to MAPI_LOGON_UI))
USE BUILTIN(MAIL_SET_OPTION) WITH_ARGS('MAPI_PASSWORD_UI')
***** COMMENT(Set Recipient using TO argument)
```

```

***** COMMENT(may also set others for TO CC BCC)
USE BUILTIN(MAIL_ADD_RECIPIENT) WITH_ARGS(TO '<-- name --
>' 'SMTP:<-- recipient@address -->') TO_GET(#LEM_RETC)
***** COMMENT(To define attachment file remove comments)
*** USE BUILTIN(MAIL_ADD_ATTACHMENT) WITH_ARGS('c:\config.s
ELSE
***** COMMENT(Set Recipient using TO argument)
USE BUILTIN(MAIL_ADD_RECIPIENT) WITH_ARGS(TO 'SMTP:<-
- recipient@address -->') TO_GET(#LEM_RETC)
ENDIF
IF COND(*HOST)
***** COMMENT(Define Message Originator if IBM i)
***** COMMENT(MAPI on non-IBM i usually defaults)
USE BUILTIN(MAIL_ADD_ORIGINATOR) WITH_ARGS('SMTP:<-
- recipient@address -->') TO_GET(#LEM_RETC)
ENDIF
***** COMMENT(Set receipt acknowledgement is required)
USE BUILTIN(MAIL_SET_OPTION) WITH_ARGS(RECEIPT_REQUESTE
***** COMMENT(Set Subject text)
USE BUILTIN(MAIL_SET_SUBJECT) WITH_ARGS('Testing Email Built-
In Functions') TO_GET(#LEM_RETC)
***** COMMENT(Set message Text)
USE BUILTIN(MAIL_ADD_TEXT) WITH_ARGS('Hello,' Y) TO_GET(#LE
DEFINE FIELD(#BIGLINE) TYPE(*CHAR) LENGTH(255) LABEL('Big Te
CHANGE FIELD(#BIGLINE) TO("I am sending this message just to try out t
In Functions.")
USE BUILTIN(MAIL_ADD_TEXT) WITH_ARGS(#BIGLINE Y) TO_GET(
USE BUILTIN(MAIL_ADD_TEXT) WITH_ARGS('Thank You' N) TO_GET(
***** COMMENT(Prompt to SEND the mail then send)
DEFINE FIELD(#LTEXT2) TYPE(*CHAR) LENGTH(1) DESC('Press ENTE
POP_UP FIELDS((#LTEXT2 *L4 *P4 *DESC *NC)) IDENTIFY(*LABEL) /
USE BUILTIN(MAIL_SEND) TO_GET(#LEM_RETC)
IF COND(*OKAY)
MESSAGE MSGTXT('Send SUCCESSFUL.')
ELSE
MESSAGE MSGTXT('Send FAILED.')
ENDIF
END_LOOP
*****

```


9.163 MAKE_FILE_OPERATIONNL

⇒ **Note:** Built-In Function Rules.

Submits a job to create or recreate a file plus associated logical files and I/O module.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition. A job is submitted to perform the operation compile.
Visual LANSAs for Windows	YES	Calls the code generator and compiler directly, and only returns when the operation is complete. Does not submit a job.
Visual LANSAs for Linux	NO	

Arguments for Visual LANSAs

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	File name	1	10		
2	A	Req	Library name	1	10		
3	A	Opt	Recreate options if file is already created Byte 1: Y = rebuild the table and save data N = do not rebuild the table Default: Y Byte 2:	1	3		

			Y = rebuild indexes/views N = do not rebuild indexes/views Default: Y Byte 3: Y = rebuild OAM N = do not rebuild OAM Default: Y				
4	A	Opt	Name of job Ignored	1	10		
5	A	Opt	Name of job description Ignored	1	21		
6	A	Opt	Name of job queue Ignored	1	21		
7	A	Opt	Name of output queue Ignored	1	21		
8	A	Opt	Produce file and I/O module source listings ? Y = keep source N = do not keep source	1	1		
9	A	Opt	Ignore decimal data error in associated I/O module? Ignored	1	1		
10	A	Opt	Strip debug data options in associated I/O module? Ignored	1	1		
11	A	Opt	User program to call Ignored	1	21		
12	A	Opt	Delete \$\$ File? (Used for Keep Saved Data) Y = Keep saved data. N = Do not keep saved data.	1	1		

			Default: N				
--	--	--	------------	--	--	--	--

Arguments for LANSA for i

For further information, refer to *Create/Recreate a file from its definition* screen in [Submitting Job to Make File Definition Operational](#) in the *LANSA for i User Guide*.

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	File name	1	10		
2	A	Req	Library name	1	10		
3	A	Opt	Recreate options if file is already created Byte 1: Y = recreate the table N = do not recreate the table Default: Y Byte 2: Y = recreate indexes N = do not recreate indexes Default: Y Byte 3: Y = rebuild OAM N = do not rebuild OAM Default: Y	1	3		
4	A	Opt	Name of batch job Default: File name	1	10		
5	A	Opt	Name of job description Default: the job description from the requesting job's attributes.	1	21		
6	A	Opt	Name of job queue	1	21		

			Default: the job queue from the requesting job's attributes.				
7	A	Opt	Name of output queue Default: the output queue from the requesting job's attributes.	1	21		
8	A	Opt	Produce file and I/O module source listings ? Y = produce listings N = do not produce listings Default N (do not produce listings)	1	1		
9	A	Opt	Ignore decimal data error in associated I/O module? Y = ignore decimal data errors N = do not ignore errors Default: N (do not ignore errors)	1	1		
10	A	Opt	Strip debug data options in associated I/O module? Y = debugging information should be stripped. N = debugging information should not be stripped. Default: Y (debugging information should be stripped)	1	1		
11	A	Opt	User program to call Default: Blank	1	21		
12	A	Opt	Delete \$\$ File? Y = \$\$ version of file should be deleted. N = \$\$ version of file should not be deleted. Default: N (\$\$ version of file should not be deleted)	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = successful submission ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued automatically.	2	2		

Example

A user wants to control the compilation of files and associated logical views and I/O module using their own version of the "Create / Re-Crete a File" facility.

```
FUNCTION OPTIONS(*DIRECT)
```

```
***** Define arguments and lists
```

```
DEFINE FIELD(#FILNAM) TYPE(*CHAR) LENGTH(010)
```

```
DEFINE FIELD(#LIBNAM) TYPE(*CHAR) LENGTH(010)
```

```
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(002)
```

```
BEGIN_LOOP
```

```
***** Request File and library name
```

```
REQUEST FIELDS((#FILNAM)(#LIBNAM))
```

```
USE BUILTIN(START_FILE_EDIT) WITH_ARGS(#FILNAM  
#LIBNAM LAN 'SALES RESULTS' NORMAL) TO_GET(#RETCOD)
```

```
USE BUILTIN(END_FILE_EDIT) WITH_ARGS(Y)  
TO_GET(#RETCOD)
```

```
***** Execute Built-In Function - MAKE_FILE_OPERATIONL
```

```
USE BUILTIN(MAKE_FILE_OPERATIONL) WITH_ARGS(#FILNAM  
#LIBNAM) TO_GET(#RETCOD)
```

```
***** Check if submission was successful
```

```
IF COND('#RETCOD *EQ "OK"')
```

```
MESSAGE MSGTXT('Create/recreate of file submitted successful')
CHANGE FIELD(#FILNAM) TO(*BLANK)
ELSE
MESSAGE MSGTXT('Create/recreate submit failed with errors, refer to
additional messages')
ENDIF
END_LOOP
```

9.164 MAKE_SOUND

⇒ **Note:** Built-In Function Rules.

Causes a standard sound to be queued.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Sound identifier. The value specified should be one of : 'BEEP' - Standard beep using the computer speaker 'ASTERISK' - System configured Asterisk sound. 'EXCLAMATION' - System configured Exclamation sound. 'HAND' - System configured Hand sound. 'QUESTION' - System - configured Question sound. 'DEFAULT' - System configured Default sound. If this argument is not specified or is specified incorrectly the default value 'BEEP' will be used.	1	20		

Return Values

There are no Return Values.

Technical Notes

- After queuing the sound the MAKE_SOUND Built-In Function returns control to the calling function. The sound will be played asynchronously.
- If the system cannot play the specified sound it attempts to play the system default sound. If it cannot play the system default sound, the function produces a standard beep sound through the computer speaker.

Example

This example is for all possible sounds as not all are available with every PC. The actual sound you hear will vary from PC to PC. (The default sound BEEP is particularly recalcitrant.)

```
function options(*DIRECT)
begin_loop
change field(#STD_TEXT) to(ASTERISK)
use builtin(MAKE_SOUND) with_args(ASTERISK)
request fields(#STD_TEXT)
change field(#STD_TEXT) to(EXCLAMATION)
use builtin(MAKE_SOUND) with_args(EXCLAMATION)
request fields(#STD_TEXT)
change field(#STD_TEXT) to(BEEP)
use builtin(MAKE_SOUND) with_args(BEEP)
request fields(#STD_TEXT)
change field(#STD_TEXT) to(HAND)
use builtin(MAKE_SOUND) with_args(HAND)
request fields(#STD_TEXT)
change field(#STD_TEXT) to(QUESTION)
use builtin(MAKE_SOUND) with_args(QUESTION)
request fields(#STD_TEXT)
change field(#STD_TEXT) to(DEFAULT)
use builtin(MAKE_SOUND) with_args(DEFAULT)
request fields(#STD_TEXT)
end_loop
```

9.165 MESSAGE_BOX_ADD

⇒ **Note:** Built-In Function Rules.

Adds one or more items to the message box assembly area as a new message line. These items would normally be displayed by the MESSAGE_BOX_SHOW, Built-In Function at some later time.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	X	Req	First item to be added to the message box assembly area	1	256	0	9
2 -10	X	Opt	Subsequent items to be added to the message box assembly area	1	256	0	9

Return Values

There are no Return Values

Technical Notes

- The first item in the list of items specified causes a new message line to be started.
- Subsequent items in the list are separated by a single blank from the preceding item.

- Items are converted to left aligned strings and trailing blanks are truncated. A completely blank/null item is converted to a single blank.
- The maximum length of 256 characters is true if the argument is specified as a character **variable**. However, the maximum length of a character **literal** is 48 characters.
- Messages from successive MESSAGE_BOX_ADD commands are added to the message box assembly area. The message box message assembly area can contain at most 4096 characters (including automatically inserted blanks or control characters). Once the assembly area reaches its maximum size all in progress or subsequent add or append operations are ignored. No notification or error is given when this situation arises.
- The message assembly area is cleared at completion of a MESSAGE_BOX_SHOW command.

Examples

Refer to [9.168 MESSAGE_BOX_SHOW](#).

9.166 MESSAGE_BOX_APPEND

⇒ **Note:** Built-In Function Rules.

Appends one or more items to the message box assembly area. A new message line is not started. These items would normally be displayed by the MESSAGE_BOX_SHOW, Built-In Function at some later time.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	X	Req	First item to be appended to the message box assembly area	1	256	0	9
2 - 10	X	Opt	Subsequent items to be appended to the message box assembly area	1	256	0	9

Return Values

There are no Return Values

Technical Notes

- This Built-In Function does not cause a new message line to be started. It is designed to append information to existing message lines.
- All items (including the first one) are separated, by a single blank from the preceding item.

- Items are converted to left aligned strings and trailing blanks are truncated. A completely blank/null item is converted to a single blank.
- The message box message assembly area can contain at most 4096 characters (including any automatically inserted blanks or control characters). Once the assembly area reaches its maximum size all in progress or subsequent add or append operations are ignored. No notification or error is given when this situation arises.

Examples

Refer to [9.168 MESSAGE_BOX_SHOW](#)

9.167 MESSAGE_BOX_CLEAR

⇒ **Note:** Built-In Function Rules.

Clears the current message box message assembly area.

For use with

LANSA for i NO

Visual LANSA for Windows YES

Visual LANSA for Linux NO

Arguments

There are no Arguments

Return Values

There are no Return Values

Technical Notes

- Use of this Built-In Function is rare because the message assembly area is automatically cleared by using MESSAGE_BOX_SHOW, Built-In Function.

Examples

Refer to [9.168 MESSAGE_BOX_SHOW](#)

9.168 MESSAGE_BOX_SHOW

⇒ **Note:** Built-In Function Rules.

Causes a standard MS Windows message box to be displayed. It then waits until the user clicks one of a specified set of buttons in the message box before returning control to the application.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Specifies the button(s) to be displayed in the message box. This value should be specified as: 'ABORTRETRYIGNORE' 'OK' 'OKCANCEL' 'RETRYCANCEL' 'YESNO' 'YESNOCANCEL' If this argument is not specified or is invalid it will be set to 'OK'.	1	20		
2	A	Opt	Specifies which of the enabled buttons is to be the default button. This value should be specified as: 'ABORT' 'RETRY'	1	20		

			<p>'IGNORE'</p> <p>'OK'</p> <p>'CANCEL'</p> <p>'YES'</p> <p>'NO'</p> <p>If this argument is not specified or is invalid it will be set to the first button specified by argument 1.</p> <p>If argument 1 is also not specified or invalid it this argument will be set to OK.</p>				
3	A	Opt	<p>Icon to be displayed in message box. This value should be specified as:</p> <p>'EXCLAMATION'</p> <p>'WARNING'</p> <p>'INFORMATION'</p> <p>'ASTERISK'</p> <p>'QUESTION'</p> <p>'STOP'</p> <p>'ERROR'</p> <p>'HAND'</p> <p>'NONE'</p> <p>If this argument is not specified or it is invalid this argument will be set to 'NONE'.</p>	1	20		
4	u	Opt	<p>Message Box Title. If this argument is not specified this argument defaults to the description of the application that is using the Built-In Function.</p>	1	256		
5	U	Opt	<p>Overriding message text. This value will override and replace any text currently in the message box assembly area. This argument allows simple messages to be presented without having to use using the ADD and APPEND Built-In Functions.</p>	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	<p>Button used to close the message box window. This will be returned as: 'ABORT' 'RETRY' 'IGNORE' 'OK' 'CANCEL' 'YES' 'NO'</p> <p>The exception to this rule is when this Built-In Function is used in a non-Windows environment. In this case the return value is the default value specified by argument 2 (regardless of its validity).</p>	1	20		

Technical Notes

- The message assembly area is cleared at completion of this Built-In Function.
- If invoked in an appropriate environment (eg: batch task, non-Windows environment) no message window is displayed and the default button value is returned. The value returned is the default value specified in argument 2.
- You should ensure that the default button, when specified, is valid and appropriate for all situations.

Examples

Present the simplest possible message box:

```
USE BUILTIN(MESSAGE_BOX_SHOW)
```

Present a message box enabling the OK and Cancel buttons (with Cancel as the default), title "HELLO", the question icon and text "Do you want to do your backups now?".

```
USE BUILTIN(MESSAGE_BOX_SHOW) WITH_ARGS(OKCANCEL CAN
```

```
if '#std_codel *eq CANCEL'  
Return  
endif
```

Format some details of an employee into the message box assembly area as individual lines and then display them with just an OK button.

```
use message_box_add ('Employee Number:' #empno)  
use message_box_add ('Name:' #givename #surname)  
use message_box_add ('Department:' #deptment)  
use message_box_add ('Section:' #section)  
use message_box_add ('Salary:' #salary '(monthly =' #mnthsal ')')  
use message_box_show
```

9.169 MESSAGE_COLLECTOR

⇒ **Note:** Built-In Function Rules.

Nominates a function as a "message collector".

For use with

LANSA for i YES Not available for RDMLX.

Visual LANSA for Windows NO

Visual LANSA for Linux NO

Arguments

No Argument Values.

Return Values

No Return Values

Technical Notes

- This Built-In Function allows a function to nominate itself as a "message collector". This means that most messages that it causes to be subsequently issued (from functions that it calls, I/O modules it invokes, or triggers it fires) will be routed **directly** to it. Normally they would be routed up the invocation stack as each object involved completes execution.
- This Built-In Function is a definition function. Its presence **anywhere** in a function causes the function to be added to the collector stack at entry, and removed at termination.

Because the MESSAGE_COLLECTOR Built-In Function is a definition function, not an executable function, you cannot make code like this work:

```
if ( ..... )  
  
use MESSAGE_COLLECTOR  
  
endif
```

This code will become a message collector regardless of the IF condition.

- Normally a USE MESSAGE_COLLECTOR command would immediately follow a FUNCTION command at the beginning of the function.
- This Built-In Function has been designed to be used in interactive functions and batch functions that are "sitting at the top" of a very complex and "deep" series of function calls and triggers. It is not usually required in mainstream applications.
- This Built-In Function has been designed to speed up this special type of application, **not** to alter its processing logic or architecture in any way.

Warnings:

- **Do not under any circumstances whatsoever** design applications that use the MESSAGE_COLLECTOR Built-In Function to affect the way that the application works. In other words, the application should be functionally identical regardless of whether this Built-In Function is used or not. MESSAGE_COLLECTOR must only be used to speed up message routing - not to implement any other form of logic or architecture that relies on its existence to work correctly.

While it is thought that designers could not use MESSAGE_COLLECTOR to change processing in any way, it is better that this important point is noted. The message routing logic used by Visual LANSA is much more efficient than the IBM i message routing architecture, and has no need of this special option to speed it up.

- Up to 10 message collectors may be stacked.

9.170 NUMERIC_STRING

⇒ **Note:** Built-In Function Rules.

Converts a number to a string.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Number to be converted	1	29	0	9
2	A	Opt	Removes trailing zeroes from the decimal portion of the number. Y = Trim Default = Trim	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Number as a string	1	31		

Example

Convert a number to a string for use in QRYSLT of OPEN command

```
DEFINE FIELD(#SALARY) TYPE(*DEC) LENGTH(20) DECIMALS(0)
DEFINE FIELD(#SALCHR) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#QRYSLT) TYPE(*CHAR) LENGTH(256)
*****
REQUEST FIELDS(#SALARY)
USE BUILTIN(NUMERIC_STRING) WITH_ARGS(#SALARY)
   TO_GET(#SALCHR)
CHANGE FIELD(#QRYSLT) TO("SALARY *GT")
USE BUILTIN(BCONCAT) WITH_ARGS(#QRYSLT #SALCHR)
   TO_GET(#QRYSLT)
OPEN FILE(PAYROLL) USE_OPTION(*OPNQRYF) QRYSLT(#QRYSLT)
```

< some processing >

9.171 OBJECT_PROPAGATE

⇒ **Note:** Built-In Function Rules.

Propagates an object to a given repository group.

For use with

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

LANSA for i	YES	Cannot be used in an RDMLX function. Can work for RDML and RDMLX objects in an RDMLX partition. Can work for RDML objects in an RDML partition.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object Name For type: Object name is Field: field name File: file name Function : process name Process: process name System variable: first 10 characters of system variable name Multilingual variable: first 10 characters of multilingual variable name	1	10	0	0
2	A	Req	Object extension.	1	10	0	0

			For type : Object extension is Field : blank File : file library name Function : function name Process : blank or partition module library System variable: last 10 characters of system variable name Multilingual variable : last 10 characters of multilingual variable name				
3	A	Req	Object type. Valid object types are: DF = Field FD= File PF =Function PD=Process SV= System variable MT= Multilingual text variable				
4	A	Req	Work Group *ALL or a valid work group				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code Valid return codes: OK= Successful ER= Not Successful	1	2	0	0

Technical Notes

The Partition Id is the current partition in which this Built-In Function is

executed.

Example

This RDML code shows how the Built-In Function can be used in different environments.

These functions can be invoked directly by using the LANSAs command with these parameters:

```
LANSAs REQUEST(RUN) PROCESS(Process_name) FUNCTION(Function_
```

```
FUNCTION *DIRECT
DEFINE FIELD(#OBJNM) TYPE(*CHAR) LENGTH(10) DESC('Object I
)
DEFINE FIELD(#OBJEX) TYPE(*CHAR) LENGTH(10) DESC('Object E
DEFINE FIELD(#OBJTY) TYPE(*CHAR) LENGTH(2) DESC('Object Ty
DEFINE FIELD(#WRKGP) TYPE(*CHAR) LENGTH(10) DESC('WorkG
DEFINE FIELD(#TSKID) TYPE(*CHAR) LENGTH(10) DESC('Task ID')
DEFINE FIELD(#RTNCD) TYPE(*CHAR) LENGTH(2) DESC('Return C
GROUP_BY NAME(#PARM_GRP) FIELDS((#OBJNM) (#OBJEX) (#OBJ
*****
BEGIN_LOOP
REQUEST FIELDS((#PARM_GRP))
USE BUILTIN(OBJECT_PROPAGATE) WITH_ARGS(#OBJNM #OBJE
END_LOOP
*
FUNCTION *DIRECT
DEFINE FIELD(#OBJNM) TYPE(*CHAR) LENGTH(10) DESC('Object N
DEFINE FIELD(#OBJEX) TYPE(*CHAR) LENGTH(10) DESC('Object E
DEFINE FIELD(#OBJTY) TYPE(*CHAR) LENGTH(2) DESC('Object Ty
DEFINE FIELD(#WRKGP) TYPE(*CHAR) LENGTH(10) DESC('WorkGr
DEFINE FIELD(#TSKID) TYPE(*CHAR) LENGTH(10) DESC('Task ID')
DEFINE FIELD(#RTNCD) TYPE(*CHAR) LENGTH(2) DESC('Return Cc
DEFINE FIELD(#MSGDTA1) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
CHANGE FIELD(#MSGDTA1) TO('RETURN CODE--> ')
GROUP_BY NAME(#PARM_GRP) FIELDS((#OBJNM) (#OBJEX) (#OBJT
IF COND(*JOBMODE = I')
BEGIN_LOOP
REQUEST FIELDS((#PARM_GRP))
```

```
SUBMIT PROCESS(TESTBIF) FUNCTION(BIF02) EXCHANGE(#PARM
CHANGE FIELD(#PARM_GRP) TO(*NULL)
END_LOOP
ELSE
USE BUILTIN(OBJECT_PROPAGATE) WITH_ARGS(#OBJNM #OBJE
*****
USE BUILTIN(BCONCAT) WITH_ARGS(#MSGDTA1 #RTNCD #OBJT
MESSAGE MSGID(CPF9898) MSGF(QCPFMSG) MSGDTA(#MSGDT
ENDIF
```

9.172 PACKAGE_BUILD

⇒ **Note:** Built-In Function Rules..

This Built-In Function will build a Version or Patch has been defined using the *Deployment Tool* or the [9.173 PACKAGE_CREATE](#) Built-In Function.

For use with

LANSA for i	NO	
Visual LANSA for Windows	YES	Only available for RDMLX.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Application Name	1	8		
2	A	Req	Version or Patch Version must be in the format <major version number>.<minor version number>.<build number> Patch must be in the format <major version number>.<minor version number>.<build number>.<patch number>	1	23		
3	A	Opt	Package Path Indicate where the package should be created. If path is blank, package is expected to exist in <System Directory>\X_Apps. Default: blank	1	256		
4	A	Opt	Replace Package Y or N Default: N	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code OK = Package built without error ER = An Error occurred during the package build. Check log files associated with package.	2	2		
2	A	Req	Reason Explanation accompanying error	1	100		

9.173 PACKAGE_CREATE

⇒ **Note:** Built-In Function Rules.. Deleteif applicable.

This Built-In Function creates a Package based on the supplied Deployment Tool Template. Objects can be added to the package.

Note: For WAMs and Weblets, the required Languages and Technology Services must be provided.

For use with

LANSA for i	NO	
Visual LANSA for Windows	YES	Only available for RDMLX.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Deployment Template Must have associated .ptf template file in <System Directory>\X_Apps\X_Tmpl	1	10		
2	A	Req	Application Name	1	8		
3	A	Req	Version or Patch Version must be in the format <major version number>.<minor version number>.<build number> Patch must be in the format <major version number>.<minor version number>.<build number>.<patch number>	1	23		
4	List	Req	Objects to include in Package List can be empty but must be supplied.	81	99		

			<p>From - To Description</p> <p>1 50 Object Type</p> <p>Supported object types are listed in Package Objects.</p> <p>51 70 Object Name</p> <p>71 80 Object Qualifier</p> <p>81 81 Include Data (Files & Web Components only)</p> <p>For files:</p> <p>Y - Include data</p> <p>N - Do not include data</p> <p>For web components:</p> <p>I - Input</p> <p>O - Output</p> <p>N - Not applicable</p> <p>82 82 Data processing option (files only)</p> <p>For files:</p> <p>I - Reload data ignoring any duplicates</p> <p>D - Drop existing data</p> <p>R - Reload data replacing duplicates (only available if file data included).</p> <p>83 99 Reserved for future use.</p>				
5	List	Req	<p>Web Designs</p> <p>If including WAMs or weblots in the package indicate the languages and technology services required.</p> <p>List can be empty but must be supplied.</p> <p>From - To Description</p> <p>1 50 Object Type</p> <p>WEBLANGUAGE TECHNOLOGYSERVICE</p> <p>51 60 Language / Provider</p> <p>WEBLANGUAGE : Language Code (e.g. ENG)</p>	70	70		

			<p>TECHNOLOGYSERVICE: Technology Service Provider (e.g. LANSA) 61 70 Technology Service</p> <p>WEBLANGUAGE: blank</p> <p>TECHNOLOGYSERVICE: Technology Service (e.g. XHTML)</p>				
6	A	Opt	<p>Package Path</p> <p>Indicate where the package should be created. If blank package will be created at <System Directory>\X_Apps.</p> <p>If you intend to edit the package using the Deployment Tool, the package must be created at <System Directory>\X_Apps</p> <p>Default: blank</p>	1	256		
7	A	Opt	<p>Replace Package</p> <p>Y or N</p> <p>Default: N</p>	1	1		
8	A	Opt	<p>Build Package</p> <p>Y or N</p> <p>Default: Y</p>	1	1		
9	A	Opt	<p>Package Description</p> <p>If the package already exists, the description will only be replaced if a non-blank description is supplied.</p> <p>Default: blank</p>	1	200		
10	A	Opt	<p>Application Description</p> <p>If application already exists, the description will only be replaced if non-blank description is supplied.</p> <p>Default: blank</p>	1	200		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code OK = Package created without error ER = An Error occurred during the package creation or build. Check log files associated with package.	2	2		
2	A	Req	Reason Explanation accompanying error	1	100		

Package Objects

When formatting entries for Argument 4, the *Objects to include in Package*, use the following table to determine appropriate data for each object type.

Object Type	Object Name Required	Object Qualifier Required	Include Data Required
ACTIVEX	Y	N	N
BITMAP	Y	N	N
BUSINESS OBJECT	Y	N	N
CURSOR	Y	N	N
DOTNETCOMPONENT	Y	N	N
EXTERNALRESOURCE	Y	N	N
FIELD	Y	N	N
FILE	Y	Y - Library name	Y
FORM	Y	N	N

FUNCTION	Y	Y - Process	N
ICON	Y	N	N
LANGUAGE	Y – Package Language Code Special value used to include Package Languages	N	N
MESSAGE	Y - Message ID (1-7), Language Code (8-11) Language is optional. If not supplied, all languages are included.	Y - Message File	
MESSAGEFILE	Y – Message File	Y - Language Code	N
MULTILINGUALVARIABLE	Y	N	N
PROCESS	Y	N	N
REUSABLEPART	Y	N	N
SYSTEMVARIABLE	Y	N	N
TECHNOLOGYSERVICE	Y – Provider	Y – Technology Service	N
VISUALSTYLE	Y	N	N
WEBAPPLICATIONMODULE	Y	N	N
WEBCOMPONENT	Y	N	Y
WEBLET	Y	N	N
WEB SERVICE	Y	N	N

What you cannot include with this BIF

- Non-LANSA objects – unless they are first recorded as [External Resources](#).
- Editor List or Task related objects.
- Application Template objects.

9.174 PHYSICAL_KEY

⇒ **Note:** Built-In Function Rules.

Specifies or re-specifies the name of a field that is a key of the physical file associated with the file definition being edited.

An edit session must be commenced using the START_FILE_EDIT Built-In Function prior to using PHYSICAL_KEY.

Warning: This Built-In Function cannot be used for a file of type "OTHER".

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of key field. Must have been previously specified as a field in the file by using the FILE_FIELD Built-In Function.	1	10		
2	N	Opt	Optional sequencing number. Used to sequence key fields. If not specified keys are sequenced in the same order as they are presented.	1	5	0	0

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = key defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically and the edit session ended without commitment.	2	2		

9.175 PUT_CHAR_AREA

⇒ **Note:** Built-In Function Rules.

Puts a character string into a character data area.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Value to put	1	2000		
2	A	Req	Data area name	1	10		
3	A	Opt	Library name Default: '*LIBL' When data area is *LDA or *GDA this argument is '*LIBL'.	1	10		
4	A	Opt	Unlock data area 'Y' - unlock data area 'N' - do not unlock data area Default: 'N'	1	1		
5	N	Opt	Start pos. to insert from. Default: position 1	1	5	0	0
6	N	Opt	Length to insert Default: full length	1	4	0	0

Note: Start position and length, if specified, must BOTH be provided as argument values.

Return Values

No return values.

Examples

Store a customer name #CUSNAM in a data area LASTCUST which resides in library QTEMP.

```
USE    BUILTIN(PUT_CHAR_AREA) WITH_ARGS(#CUSNAM LASTCU
```

Store a customer name #CUSNAM in bytes 101 to 140 of a data area called INFOCUST which resides in library QTEMP.

```
USE    BUILTIN(PUT_CHAR_AREA) WITH_ARGS(#CUSNAM INFOCU
```

Store user name in the *LDA data area to be used for report headings.

```
USE    BUILTIN(PUT_CHAR_AREA) WITH_ARGS("JOHN SMITH" ""*I
```

The first 10 positions of *GDA are updated by the current job. As a group job becomes active it sets a flag in the *GDA.

In other words by retrieving the *GDA you can find out what group jobs you have already activated.

```
CASE    OF_FIELD(#GROUP)
WHEN    VALUE_IS('*EQ JOB0001')
USE     BUILTIN(PUT_CHAR_AREA) WITH_ARGS('Y' ""*GDA"" ""*LIBI
WHEN    VALUE_IS('*EQ JOB0002')
USE     BUILTIN(PUT_CHAR_AREA) WITH_ARGS('Y' ""*GDA"" ""*LIBI
ENDCASE
```

9.176 PUT_COND_CHECK

⇒ **Note:** Built-In Function Rules.

Creates/amends a "simple conditional logic" DICTIONARY or FILE level validation check into the data dictionary or file definition of the nominated field.

When adding a FILE level validation check to a field, the file involved must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

All argument values passed to this Built-In Function are validated exactly as if they had been entered through the online validation check definition screen panels.

Normal authority and task tracking rules apply to the use of this Built-In Function.

For more information refer to [Field Rules and Triggers](#) in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	Validation performed on the Windows platform is not as rigorous as that performed by LANSA for i.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation check. D = Dictionary level F = File level	1	1		

2	A	Req	Name of field in dictionary to which validation rule is to be applied.	1	10		
3	N	Req	Sequence number of check.	1	3	0	0
4	A	Req	Description of check.	1	30		
5	A	Req	Enable check for ADD. Y = Check performed on ADD U = Check performed on ADDUSE N = Check not performed on ADD	1	1		
6	A	Req	Enable check for CHANGE. Y = Check performed on CHG U = Check performed on CHGUSE N = Check not performed on CHG	1	1		
7	A	Req	Enable check for DELETE. Y = Check performed on DLT N = Check not performed on DLT	1	1		
8	A	Req	Action if check is true. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
9	A	Req	Action if check is false. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
10	A	Req	Message file details. Details of error message to be issued from a message file. Message file details should be formatted as follows: From - To Description 1 - 7 Error Message Number 8 - 17 Message File Name 18 - 27 Message File Library.	27	27		

			If message text is used, pass this argument as blanks				
11	A	Req	Message text.	1	80		
12	L	Req	Working list to contain the condition that is to be evaluated for the simple logic check. The calling RDML function must provide a working list with an aggregate entry length of exactly 79 bytes and exactly 5 condition line entries. Each list entry sent should be formatted as follows: From - To Description 1 - 79 Condition line	1	20		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

Example

A user wants to put a "simple conditional logic" validation check for a specific field, without going through the LANSAs options provided on the "Field Control Menu" that enables the user to put a "simple conditional logic" validation check.

```
***** Define arguments and lists
DEFINE FIELD(#LEVEL) TYPE(*CHAR) LENGTH(1) LABEL('Level')
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(10) LABEL('Field')
```

```

DEFINE FIELD(#SEQNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0) I
DEFINE FIELD(#DESCR) TYPE(*CHAR) LENGTH(30) LABEL('Descrip
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2) LABEL('Return
DEFINE FIELD(#ENBADD) TYPE(*CHAR) LENGTH(1) LABEL('Enable
DEFINE FIELD(#ENBCHG) TYPE(*CHAR) LENGTH(1) LABEL('Enable
DEFINE FIELD(#ENBDLT) TYPE(*CHAR) LENGTH(1) LABEL('Enable
DEFINE FIELD(#TRUE) TYPE(*CHAR) LENGTH(6) LABEL('Action if
DEFINE FIELD(#FALSE) TYPE(*CHAR) LENGTH(6) LABEL('Action if
DEFINE FIELD(#MSGDET) TYPE(*CHAR) LENGTH(27) LABEL('Mess
DEFINE FIELD(#MSGTXT) TYPE(*CHAR) LENGTH(80) LABEL('Mess
DEFINE FIELD(#CONLIN) TYPE(*CHAR) LENGTH(79) LABEL('Condi
DEF_LIST NAME(#CONWRK) FIELDS((#CONLIN)) TYPE(*WORKING
DEF_LIST NAME(#CONBRW) FIELDS((#CONLIN)) ENTRYS(5)
GROUP_BY NAME(#VALCHK) FIELDS((#LEVEL) (#FIELD) (#SEQNUM
***** Initialize Browse list
CLR_LIST NAMED(#CONBRW)
INZ_LIST NAMED(#CONBRW) NUM_ENTRYS(5) WITH_MODE(*CHA
***** Clear Working lists
BEGIN_LOOP
CLR_LIST NAMED(#CONWRK)
***** Request Validation check details
REQUEST FIELDS((#VALCHK)) BROWSELIST(#CONBRW)
***** Load key field working list
SELECTLIST NAMED(#CONBRW)
ADD_ENTRY TO_LIST(#CONWRK)
ENDSELECT
***** Execute Built-In Function - PUT_COND_CHECK
USE BUILTIN(PUT_COND_CHECK) WITH_ARGS(#LEVEL #FIELD ;
***** Put "simple conditional logic" successful
IF COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Put "simple conditional logic" validation check(s) wa
***** Put "simple conditional logic" failed
ELSE
IF COND('#RETCOD *EQ "ER"')
MESSAGE MSGTXT('Put "simple conditional logic" validation check(s) fai
ENDIF
ENDIF
END_LOOP

```

9.177 PUT_DATE_CHECK

⇒ **Note:** Built-In Function Rules.

Creates/amends a "date range / date format" DICTIONARY or FILE level validation check into the data dictionary or file definition of the nominated field. When adding a FILE level validation check to a field, the file involved must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

All argument values passed to this Built-In Function are validated exactly as if they had been entered through the online validation check definition screen panels.

Normal authority and task tracking rules apply to the use of this Built-In Function.

For more information refer to *Field Rules and Triggers* in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation check. D = Dictionary level F = File level	1	1		
2	A	Req	Name of field in dictionary to which	1	10		

			validation rule is to be applied.				
3	N	Req	Sequence number of check.	1	3	0	0
4	A	Req	Description of check.	1	30		
5	A	Req	Enable check for ADD. Y = Check performed on ADD U = Check performed on ADDUSE N = Check not performed on ADD	1	1		
6	A	Req	Enable check for CHANGE. Y = Check performed on CHG U = Check performed on CHGUSE N = Check not performed on CHG	1	1		
7	A	Req	Enable check for DELETE. Y = Enable check. N = Do not enable check.	1	1		
8	A	Req	Action if check is true. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
9	A	Req	Action if check is false. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
10	A	Req	Message file details. Details of error message to be issued from a message file. Message file details should be formatted as follows: From - To Description 1 - 7 Error Message Number 8 - 17 Message File Name	27	27		

			18 - 27 Message File Library If message text is used, pass this argument as blanks.				
11	A	Req	Message text.	1	80		
12	A	Req	Format that date is to be validated in.	1	8		
13	N	Opt	Number of days allowed into the past for specified date. If not specified, a value of 9999999 is assumed.	1	7	0	0
14	N	Opt	Number of days allowed into the future for specified date. If not specified, a value of 9999999 is assumed.	1	7	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

Example

A user wants to put a "date range / date format" validation check for a specific field, without going through the LANSAs options provided on the "Field Control Menu", that enables the user to put a "date range / date format" validation check.

***** Define arguments and lists

```
DEFINE FIELD(#LEVEL) TYPE(*CHAR) LENGTH(1) LABEL('Level')
```

```

DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(10) LABEL('Field')
DEFINE FIELD(#SEQNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0) L
DEFINE FIELD(#DESCR) TYPE(*CHAR) LENGTH(30) LABEL('Descrip
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2) LABEL('Return
DEFINE FIELD(#ENBADD) TYPE(*CHAR) LENGTH(1) LABEL('Enable
DEFINE FIELD(#ENBCHG) TYPE(*CHAR) LENGTH(1) LABEL('Enable
DEFINE FIELD(#ENBDLT) TYPE(*CHAR) LENGTH(1) LABEL('Enable
DEFINE FIELD(#TRUE ) TYPE(*CHAR) LENGTH(6) LABEL('Action if
DEFINE FIELD(#FALSE) TYPE(*CHAR) LENGTH(6) LABEL('Action if
DEFINE FIELD(#MSGDET) TYPE(*CHAR) LENGTH(27) LABEL('Mess:
DEFINE FIELD(#MSGTXT) TYPE(*CHAR) LENGTH(80) LABEL('Mess:
DEFINE FIELD(#DATFMT) TYPE(*CHAR) LENGTH(8) LABEL('Date fo
DEFINE FIELD(#DAYPST) TYPE(*DEC) LENGTH(7) DECIMALS(0) LA
DEFINE FIELD(#DAYFUT) TYPE(*DEC) LENGTH(7) DECIMALS(0) LA
GROUP_BY NAME(#VALCHK) FIELDS((#LEVEL) (#FIELD) (#SEQNUM
***** Request Validation check details
BEGIN_LOOP
REQUEST FIELDS((#VALCHK))
***** Execute Built-In Function - PUT_DATE_CHECK
USE BUILTIN(PUT_DATE_CHECK) WITH_ARGS(#LEVEL #FIELD #
***** Put "date range/format" validation check was successful
IF COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Put "date range/format" validation check(s) was succe
***** Put "date range/format" failed
ELSE
IF COND('#RETCOD *EQ "ER"')
MESSAGE MSGTXT('Put "date range/format" validation check(s) failed')
ENDIF
ENDIF
END_LOOP

```

9.178 PUT_FIELD

⇒ **Note:** Built-In Function Rules.

Either inserts a new field into the LANSAs Repository or updates details of an existing field.

Optionally this Built-In Function can present a prompt screen to the user that will allow details of a new or amended field to be further specified.

Argument values are exactly as the information presented in the [Detailed Display of a Field Definition](#) in the *LANSAs for i User Guide*.

When a new field is being inserted into the dictionary, arguments that are not passed to the Built-In Function (or passed as null values) will adopt default values as described in [Creating a New Field Definition](#) in the *LANSAs for i User Guide*.

When an existing field is being updated in the Repository, arguments that are not passed to the Built-In Function (or passed as null values) will remain unchanged by the update operation.

When zero is input as the 'Number of Decimals' parameter, it is treated as a null value. Use -1 in 'Number of Decimals' parameter to indicate a request to change the number of decimals of a field to zero.

If the copy validation checks option is used, all checks from the sequence number specified are deleted, then the validation checks are copied from the 'from field'. Any reference to the 'from field' in copied validation checks are replaced by the name of the field being inserted/updated.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	The prompting option is not supported in this environment. The validation performed by this Built-In Function in the Windows environment is not as rigorous as that performed by the LANSAs in the IBM i environment
Visual	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Prompt control details Byte 1 - Prompt required Y = prompt the user N = do not prompt the user Byte 2 - EXIT/SYSTEM key Y = enable EXIT/SYSTEM key N = do not enable EXIT/SYSTEM key Byte 3 - MENU key Y = enable MENU key N = do not enable MENU key	1	3		
2	A	Req	Name of field to be inserted or updated	1	10		
3	A	Opt	Field type A = alphanumeric S = signed decimal numeric P = packed decimal numeric	1	1		
4	N	Opt	Length of field or total number of digits in field. Note: For type A must be in range 1 - 256. For type P or S must be in range 1 - 30.	3	15	0	0
5	N	Opt	Number of decimal positions Not applicable to type A field. If updating an existing field, use -1 to make this argument 0.	1	15	0	0

6	A	Opt	Reference field name If updating an existing field, use '*NONE' to remove an existing reference field name.	1	10		
7	A	Opt	Field description	1	40		
8	A	Opt	Field label	1	15		
9	A	Opt	Field column headings List of 3 * A(20) headings From - To Description 1 - 20 Column Head 1 21 - 40 Column head 2 41 - 60 Column head 3	1	60		
10	A	Opt	Output attributes list List of 10 * A(4) attributes	1	40		
11	A	Opt	Input attributes list List of 10 * A(4) attributes	1	40		
12	A	Opt	Edit code or edit word If first char is a quote (') then value is an edit word. Otherwise it is an edit code. Not applicable to type A field	1	20		
13	A	Opt	Default value of field	1	20		
14	A	Opt	Optional alias name of field	1	30		
15	A	Opt	System field flag YES = a system field NO = not a system field	3	3		
16	A	Opt	Initial public access Ignored for update operations	1	7		
17	A	Opt	Keyboard shift	1	1		
18	A	Opt	Prompting Process/Function The first 10 bytes are PROCESS name, the next 7 are FUNCTION name.	1	17		
19	A	Opt	(Re)Copy validation checks from Repository	1	10		

			field				
20	N	Opt	Starting sequence for copy	1	3	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = field inserted/updated EX = prompt was terminated by EXIT/SYSTEM function key MU = prompt was terminated by MENU/CANCEL function key ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued automatically.	2	2		

9.179 PUT_FIELD_ML

⇒ **Note:** All Multilingual Built-In Functions in [Built-In Function Rules](#).

Puts/updates a list of field multilingual attributes in different languages.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Field name	1	10		
2	L	Req	Working list to contain language code and field multilingual attributes. The function must supply a working list with an aggregate entry length of exactly 119 bytes. Each list entry sent should be formatted as follows: From - To Description 1 - 4 Language Code 5 - 44 Field description 45 - 59 Field label 60 - 79 Field column heading 1 80 - 99 Field column heading 2 100 - 119 Field column heading 3	119	119		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = multilingual attributes added / updated to the database successfully. ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued automatically.	2	2		

9.180 PUT_FILE_CHECK

⇒ **Note:** Built-In Function Rules.

Creates/amends a "code/table file lookup" DICTIONARY or FILE level validation check into the data dictionary or file definition of the nominated field.

When adding a FILE level validation check to a field, the file involved must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

All argument values passed to this Built-In Function are validated exactly as if they had been entered through the online validation check definition screen panels.

Normal authority and task tracking rules apply to the use of this Built-In Function.

For more information refer to *Field Rules and Triggers* in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation check. D = Dictionary level F = File level	1	1		
2	A	Req	Name of field in dictionary to which	1	10		

			validation rule is to be applied.				
3	N	Req	Sequence number of check.	1	3	0	0
4	A	Req	Description of check.	1	30		
5	A	Req	Enable check for ADD. Y = Check performed on ADD U = Check performed on ADDUSE N = Check not performed on ADD	1	1		
6	A	Req	Enable check for CHANGE. Y = Check performed on CHG U = Check performed on CHGUSE N = Check not performed on CHG	1	1		
7	A	Req	Enable check for DELETE. Y = Enable check. N = Do not enable check.	1	1		
8	A	Req	Action if check is true. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
9	A	Req	Action if check is false. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
10	A	Req	Message file details. Details of error message to be issued from a message file. Message file details should be formatted as follows: From - To Description 1 - 7 Error Message Number	27	27		

			8 - 17 Message File Name 18 - 27 Message File Library If message text is used, pass this argument as blanks.				
11	A	Req	Message text.	1	80		
12	A	Req	Name of file that check is to be performed against.	1	10		
13	L	Req	Working list to contain key fields/values to use when checking in the file. The calling RDML function must provide a working list with an aggregate entry length of exactly 20 bytes and at most 10 key fields/values entries may be specified. Each list entry sent should be formatted as follows : From - To Description 1 - 20 Key fields/values	1	20		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

Example

A user wants to put a "code/table file lookup" validation check for a specific field, without going through the LANSAs options provided on the "Field Control Menu" that enables the user to put a "code / table file lookup" validation check.

```
***** Define arguments and lists
DEFINE FIELD(#FILNAM) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#LIBNAM) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#LEVEL) TYPE(*CHAR) LENGTH(1) LABEL('Level')
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(10) LABEL('Field')
DEFINE FIELD(#SEQNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0) LABEL('Seq Num')
DEFINE FIELD(#DESCR) TYPE(*CHAR) LENGTH(30) LABEL('Description')
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2) LABEL('Return Code')
DEFINE FIELD(#ENBADD) TYPE(*CHAR) LENGTH(1) LABEL('Enable Add')
DEFINE FIELD(#ENBCHG) TYPE(*CHAR) LENGTH(1) LABEL('Enable Change')
DEFINE FIELD(#ENBDLT) TYPE(*CHAR) LENGTH(1) LABEL('Enable Delete')
DEFINE FIELD(#TRUE) TYPE(*CHAR) LENGTH(6) LABEL('Action if True')
DEFINE FIELD(#FALSE) TYPE(*CHAR) LENGTH(6) LABEL('Action if False')
DEFINE FIELD(#MSGDET) TYPE(*CHAR) LENGTH(27) LABEL('Message Detail')
DEFINE FIELD(#MSGTXT) TYPE(*CHAR) LENGTH(80) LABEL('Message Text')
DEFINE FIELD(#CODFIL) TYPE(*CHAR) LENGTH(10) LABEL('File Name')
DEFINE FIELD(#KEYFLD) TYPE(*CHAR) LENGTH(20) LABEL('Key Field')
DEF_LIST NAME(#KEYWRK) FIELDS((#KEYFLD)) TYPE(*WORKING)
DEF_LIST NAME(#KEYBRW) FIELDS((#KEYFLD)) ENTRIES(10)
GROUP_BY NAME(#VALCHK) FIELDS((#LEVEL) (#FIELD) (#SEQNUM))
***** Initialize Browse list
CLR_LIST NAMED(#KEYBRW)
INZ_LIST NAMED(#KEYBRW) NUM_ENTRIES(10) WITH_MODE(*CHANGED)
***** Start file edit
REQUEST FIELDS(#FILNAM #LIBNAM)
*****
USE BUILTIN(START_FILE_EDIT) WITH_ARGS(#FILNAM #LIBNAM)
***** Clear Working lists
BEGIN_LOOP
CLR_LIST NAMED(#KEYWRK)
***** Request Validation check details
REQUEST FIELDS((#VALCHK)) BROWSELIST(#KEYBRW)
***** Load key field working list
SELECTLIST NAMED(#KEYBRW)
ADD_ENTRY TO_LIST(#KEYWRK)
```

```
ENDSELECT
***** Execute Built-In Function - PUT_FILE_CHECK
USE      BUILTIN(PUT_FILE_CHECK) WITH_ARGS(#LEVEL #FIELD #S
***** Put "code/table file lookup" validation successful
IF      COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Put "code/table file lookup" validation check(s) was s
***** Put "code/table file lookup" failed
ELSE
IF      COND('#RETCOD *EQ "ER"')
MESSAGE MSGTXT('Put "code/table file lookup" validation check(s) failed
ENDIF
ENDIF
END_LOOP
USE      BUILTIN(END_FILE_EDIT) ('Y')
```

9.181 PUT_FILE_ML

⇒ **Note:** All Multilingual Built-In Functions in [Built-In Function Rules](#).

Puts/updates a list of file multilingual attributes in different languages.

An edit session must be commenced by using the START_FILE_EDIT Built-In Function prior to using this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Physical file or logical view name.	1	10		
2	L	Req	Working list to contain language code and file multilingual attributes. The function must supply a working list with an aggregate entry length of exactly 44 bytes. Each list entry sent should be formatted as follows: From - To Description 1 - 4 Language code 5 - 44 Physical file or logical view description	44	44		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = multilingual attributes added / updated to the database successfully. ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued automatically.	2	2		

9.182 PUT_FUNCTION_ATTR

⇒ **Note:** Built-In Function Rules.

Sets an attribute of a function definition that is being edited within an edit session previously started by using the START_FUNCTION_EDIT Built-In Function.

Attributes set or returned by this Built-In Function have the same editing and validation rules as the equivalent online facility provided in a full LANSAs development environment.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of attribute to change Valid attribute names are: DESC- Function description ONMENU - Display on Menu MENUSQ - Menu sequence Number	1	10		

			EDTSRC - Identifier of Editor				
2	A	Req	<p>Value of that attribute is to be changed to.</p> <p>Allowable values are as follows</p> <p>For attribute DESC: Any valid new function description up to 40 characters in length</p> <p>For attribute ONMENU: Y – Displayed on Menu N – Not displayed on Menu</p> <p>For attribute MENUSQ: Valid number represented as 5 numbers in range 00001 to 99999.</p> <p>For attribute EDTSRC: 3 character editing "source" identifier. Must not be blank or LAN.</p>	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return code</p> <p>OK = operation completed</p> <p>ER = fatal error detected</p>	2	2		

9.183 PUT_FUNCTION_ML

⇒ **Note:** All Multilingual Built-In Functions in [Built-In Function Rules](#).

Puts/updates a list of function multilingual attributes in different languages.

An edit session must be commenced by using the START_FUNCTION_EDIT Built-In Function prior to using this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain language code and function multilingual attributes. The function must supply a working list with an aggregate entry length of exactly 44 bytes. Each list entry sent should be formatted as follows: From - To Description 1 - 4 Language code 5 - 44 Function description	44	44		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = multilingual attributes added / updated to the database successfully. ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued automatically.	2	2		

9.184 PUT_FUNCTION_RDML

⇒ **Note:** Built-In Function Rules.

Stores the RDML code associated with a function from a working list.

This Built-In Function can only be used against a function that has been previously placed into an edit session using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list Name If the edit session is Include RDML audit stamps N, then the working list must have an aggregate entry length of 72 bytes where each entry is composed of: From - To Description 1 - 4 Command sequence number Signed(4, 0) 5 - 7 Command Label A(3) 8 - 17 Command A(10).				

			<p>19 - 72 Command Parameters A(55).</p> <p>If the edit session is Include RDML audit stamps Y, then the working list must have an aggregate entry length of 99 bytes, where, in addition to the positions described for Include RDML audit stamps N, each entry is composed of:</p> <p>From - To Description</p> <p>73 - 73 Command Changed Flag, to be set to Y if the RDML command was added or changed in this edit session, N otherwise.A(1)</p> <p>74 - 81 Command Changed Date S(8, 0). (CCYYMMDD). Must never be changed or set.</p> <p>82 - 91 Command Changed User. Must never be changed or set. A(10).</p> <p>92 - 99 Command Changed Task. Must never be changed or set. A(8).</p>				
2	A	Req	<p>Nominated Editing Source Must not be blank or LAN. Used to "tag" edited RDML with last editor identifier.</p> <p>*S=Signed</p>	3	3		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return code</p> <p>OK = operation completed</p> <p>ER = fatal error detected</p>	2	2		

Technical Notes

If the function is being created by this edit session and the edit session is

Include RDML audit stamps Y, the value of the Command Changed Flag is ignored.

Commands that have more than 55 bytes of parameters must be formatted across multiple entries as shown in this example:

Seq	Lab	Command	Parameters
0001		*****	This is a comment line
0002		SET_MODE	TO(*CHANGE)
0003	L32	GROUP_BY	NAME(#GROUP) FIELDS(#FIELD001 #FIELD002
0003			#FIELD003 #FIELD004 #FIELD005 #FIELD006)
0004		DISPLAY	FIELDS(#GROUP)
0005		MENU	
0006		*****	This is a comment line

9.185 PUT_HELP

⇒ **Note:** Built-In Function Rules.

Puts/updates a list of help text for a specified field, function or process.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object name The name of a field, function or process.	1	10		
2	A	Req	Object extension name If the object type is a function then this value should contain the name of the process in which the function is defined. If the object type is not a function then this value should be blank.	1	10		
3	A	Req	Object type Values: DF - Field PD - Process PF - Function	2	2		
4	L	Req	Working list to contain help text. The calling	1	77		

		<p>RDML function must provide a working list with an aggregate entry length of exactly 77 bytes.</p> <p>Each list entry sent should be formatted as follows:</p> <p>From - To Description 1 - 77 Help Text</p>				
--	--	--	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return code</p> <p>OK = help text put/updated to database successfully.</p> <p>ER = argument details are invalid or an authority problem has occurred.</p> <p>In case of "ER" return code error message(s) are issued automatically.</p>	2	2		

Example

A user wants to retrieve and update the help text of a specific object without going through the LANSAs options provided on the "Process Control Menu" and the "Field Control Menu", that enables the user to create/change help text for fields, functions and processes.

```

***** Define arguments and lists
DEFINE FIELD(#OBJNAM) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#OBJEXT) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#OBJTYP) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#HLPTXT) TYPE(*CHAR) LENGTH(77)
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
DEF_LIST NAME(#WKHLPL) FIELDS((#HLPTXT)) TYPE(*WORKING

```

```

DEF_LIST  NAME(#BWHLPL) FIELDS((#HLPTXT))
GROUP_BY  NAME(#RQSOBJ) FIELDS((#OBJNAM) (#OBJEXT) (#OBJ
GROUP_BY  NAME(#DSPHLP) FIELDS((#OBJNAM) (#OBJEXT) (#OBJ
***** Clear working and browse lists
BEGIN_LOOP
CLR_LIST  NAMED(#WKHLPL)
CLR_LIST  NAMED(#BWHLPL)
***** Request Object Name, Extension and Type
REQUEST   FIELDS(#RQSOBJ)
***** Execute Built-In Function - GET_HELP
USE       BUILTIN(GET_HELP) WITH_ARGS(#OBJNAM #OBJEXT #OBJ
***** Help text was retrieved successfully
IF        COND('#RETCOD *EQ "OK"')
***** Move Help text from the working list to the browselist
SELECTLIST NAMED(#WKHLPL)
ADD_ENTRY  TO_LIST(#BWHLPL) WITH_MODE(*CHANGE)
ENDSELECT
***** Allow Help text to be changed for the object
REQUEST   FIELDS(#DSPHLP) BROWSELIST(#BWHLPL)
***** Change the help text for this object
EXECUTE   SUBROUTINE(PUTHELP)
***** Working list overflowed, more help text to retrieve
ELSE
IF        COND('#RETCOD *EQ "OV"')
MESSAGE   MSGTXT('List not big enough to fit all help text')
***** GET_HELP failed with errors, report error
ELSE
MESSAGE   MSGTXT('GET_HELP failed with errors, try again')
ENDIF
ENDIF
END_LOOP
***** Subroutine to change help text for this object
SUBROUTINE NAME(PUTHELP)
CLR_LIST  NAMED(#WKHLPL)
***** Move Help text from the browselist to the working list
SELECTLIST NAMED(#BWHLPL)
ADD_ENTRY  TO_LIST(#WKHLPL)
ENDSELECT
***** Execute Built-In Function - PUT_HELP

```

```
USE      BUILTIN(PUT_HELP) WITH_ARGS(#OBJNAM #OBJEXT #OB.
*****  Help text was changed successfully
IF      COND(#RETCOD *EQ "OK")
MESSAGE MSGTXT('Help text for this object has been changed')
*****  PUT_HELP failed with errors, report error
ELSE
MESSAGE MSGTXT('PUT_HELP failed with errors, try again')
ENDIF
ENDROUTINE
```

9.186 PUT_ML_VARIABLE

⇒ **Note:** All Multilingual Built-In Functions in [Built-In Function Rules](#).

Adds/Updates a multilingual variable definition. to the Repository.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Multilingual variable name	5	20		
2	N	Req	Length / Total digits	1	3	0	0
3	L	Req	Working list to contain multilingual definition information. RDML An RDML list must be formatted with an aggregate entry length of exactly 82 bytes. Bytes 1-4: Language code Bytes 5-82: Multilingual variable value. RDMLX An RDMLX list must be formatted as: Alpha (4): Language code NChar (39): Multilingual variable value.	82	82		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code (OK, ER)	2	2		

9.187 PUT_NUM_AREA

⇒ **Note:** Built-In Function Rules.

Puts a numeric value into a numeric data area.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Value to put	1	15	0	0
2	A	Req	Data area name	1	10		
3	A	Opt	Library name Default: '*LIBL'	1	10		
4	A	Opt	Unlock data area 'Y' - unlock data area 'N' - do not unlock data area Default: 'N'	1	1		

Return Values

No return values.

Example

Retrieve a batch number #BATCH from data area named NEXTBATCH which should be located via the job's library list.

Increment the batch number value and place the incremented value back into the data area.

Make sure that no 2 jobs can be assigned the same batch number by using the lock and unlock options.

```
USE    BUILTIN(GET_NUM_AREA)
      WITH_ARGS(NEXTBATCH "*"LIBL" 'Y') TO_GET(#BATCH)
CHANGE FIELD(#BATCH) TO('#BATCH + 1')
USE    BUILTIN(PUT_NUM_AREA)
      WITH_ARGS(#BATCH NEXTBATCH "*"LIBL" 'Y')
```

9.188 PUT_PROCESS_ACTIONS

⇒ **Note:** Built-In Function Rules.

Puts the definition of an action bar layout into the definition of the process definition currently being edited by the START_PROCESS_EDIT Built-In Function.

Information passed into this Built-In Function is subjected to the same editing and validation rules as the equivalent online facility provided in a full LANSAs development environment.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	<p>Action Bar Definition List 1.</p> <p>This working list must contain at least 1 entry and may contain at most 18.</p> <p>For each entry in this list there must also be an entry in action bar definition list number 2.</p> <p>Lists 1 and 2 are conceptually just one list that must be passed as two real lists to get around the 256 byte list entry</p>				

			<p>length limit that LANSAs imposes.</p> <p>Each working list entry must have an aggregate length of 200 bytes and be formatted exactly as follows:</p> <p>From - To Description</p> <p>1 - 10 A(10) Action Bar Keyword</p> <p>11 - 13 A(3) AB\$OPT Code</p> <p>14 - 15 P(3,0) Number of Pull Down Options define in following array structures.</p> <p>16 - 195 A(9*20) Array of 9 x alpha 20 Pull Down Option Descriptions</p> <p>196 - 200 A(5) Spare area for future expansion of function</p>				
2	L	Req	<p>Action Bar Definition List 2</p> <p>This working list must contain at least 1 entry and may contain at most 18.</p> <p>For each entry in this list there must also be an entry in action bar definition list number 1. Lists 1 and 2 are conceptually just one list that must be passed as two real lists to get around the 256 byte list entry length limit that LANSAs imposes.</p> <p>Each working list entry must have an aggregate length of 200 bytes and be formatted exactly as follows:</p> <p>From - To Description</p> <p>1 - 18 A(9*2) Array of 9 x alpha 2 Accelerator Function Key Numbers.</p> <p>19 - 45 A(9*3) Array of 9 x alpha 3 PD\$OPT identification values.</p> <p>46 - 54 A(9*1) Array of 9 x alpha 1 initial availability flags.</p> <p>55 - 144 A(9*10) Array of 9 x alpha 10 function names. Used to indicate name of function within this process that is to be invoked.</p> <p>145 - 171 A(9*3) Array of 9 x alpha 3 process attachment sequence numbers. Used to specify the sequence number of an "attached" process or function that is to be invoked.</p> <p>172 - 200 A(29) Spare area for future expansion of function.</p>				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
----	------	-------------	-------------	------------	------------	------------	------------

1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		
---	---	-----	---	---	---	--	--

9.189 PUT_PROCESS_ATTACH

⇒ **Note:** Built-In Function Rules.

Puts a process and/or function "attachment" into the definition of the process definition currently being edited by the START_PROCESS_EDIT Built-In Function.

Information passed into this Built-In Function is subjected to the same editing and validation rules as the equivalent online facility provided in a full LANSAs development environment.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Sequencing Number.	1	3	0	0
2	A	Req	Name of process to attach.	1	10		
3	A	Req	Name of function to attach.	1	7		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.190 PUT_PROCESS_ATTR

⇒ **Note:** Built-In Function Rules.

Sets an attribute of a process definition that is being edited within an edit session previously started using the START_PROCESS_EDIT Built-In Function.

Attributes set or returned by this Built-In Function have the same editing and validation rules as the equivalent online facility provided in a full LANSAs development environment.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of attribute to change Valid attribute names are: DESC- Process Description TYPE- Process Type OPTCOM - Optimize for remote	1	10		

			comms ENAWEB - Enable for the Web ENAXML - Enable for XML Generation.				
2	A	Req	Value of that attribute is to be changed to. Allowable values are as follows: For attribute DESC: Any valid new process description up to 40 characters in length. For attribute TYPE: SAA/CUA ACT/BAR For attribute OPTCOMM: Y – Optimized for remote comms N – Not optimized for remote comms For attribute ENAWEB: Y – Enabled for the Web N – Not enabled for the Web For attribute ENAXML: Y – Enabled for XML Generation N – Not enabled for XML Generation	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.191 PUT_PROCESS_ML

⇒ **Note:** All Multilingual Built-In Functions in [Built-In Function Rules](#).

Puts/updates a list of process, multilingual attributes, in different languages.

An edit session must be commenced using the START_PROCESS_EDIT Built-In Function prior to using this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working list to contain language code and process multilingual attributes. The function must supply a working list with an aggregate entry length of exactly 44 bytes. Each list entry sent should be formatted as follows: From - To Description 1 - 4 Language code 5 - 44 Process description	44	44		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = multilingual attributes added / updated to the database successfully. ER = argument details are invalid or an authority problem has occurred. In case of "ER" return code error message(s) are issued automatically.	2	2		

9.192 PUT_PROGRAM_CHECK

⇒ **Note:** Built-In Function Rules.

Creates/amends a "call user program" DICTIONARY or FILE level validation check into the data dictionary or file definition of the nominated field.

When adding a FILE level validation check to a field, the file involved, must have been previously placed into an edit session, by the START_FILE_EDIT Built-In Function.

All argument values passed to this Built-In Function are validated exactly as if they had been entered through the online validation check definition screen panels.

Normal authority and task tracking rules apply to the use of this Built-In Function.

For more information refer to *Field Rules and Triggers* in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation check. D = Dictionary level F = File level	1	1		
2	A	Req	Name of field in dictionary to which	1	10		

			validation rule is to be applied.				
3	N	Req	Sequence number of check.	1	3	0	0
4	A	Req	Description of check.	1	30		
5	A	Req	Enable check for ADD. Y = Check performed on ADD U = Check performed on ADDUSE N = Check not performed on ADD	1	1		
6	A	Req	Enable check for CHANGE. Y = Check performed on CHG U = Check performed on CHGUSE N = Check not performed on CHG	1	1		
7	A	Req	Enable check for DELETE. Y = Enable check. N = Do not enable check.	1	1		
8	A	Req	Action if check is true. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
9	A	Req	Action if check is false. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
10	A	Req	Message file details Details of error message to be issued from a message file. Message file details should be formatted as follows: From - To Description 1 - 7 Error Message Number	27	27		

			8 - 17 Message File Name 18 - 27 Message File Library If message text is used, pass this argument as blanks.				
11	A	Req	Message text.	1	80		
12	A	Req	Name of program that is to be called to perform this check. Prefix name by "LF=" if a function is to be called. Note - additional parameters are not allowed if a function is performing the check.	1	10		
13	L	Req	Working list to contain the additional parameters that should be passed to the nominated program. The calling RDML function must provide a working list with an aggregate entry length of exactly 20 bytes and exactly 10 parameter entries. Each list entry sent should be formatted as follows: From - To Description 1 - 20 Additional parameter	1	20		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

Example

A user wants to put a "call user program" validation check for a specific field, without going through the LANSAs options provided on the "Field Control Menu" that enables the user to put a "call user program" validation check.

```
***** Define arguments and lists
DEFINE FIELD(#LEVEL) TYPE(*CHAR) LENGTH(1) LABEL('Level')
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(10) LABEL('Field')
DEFINE FIELD(#SEQNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
    LABEL('Sequence #')
DEFINE FIELD(#DESCR) TYPE(*CHAR) LENGTH(30) LABEL('Description')
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2) LABEL('Return Code')
DEFINE FIELD(#ENBADD) TYPE(*CHAR) LENGTH(1) LABEL('Enable Add')
DEFINE FIELD(#ENBCHG) TYPE(*CHAR) LENGTH(1) LABEL('Enable Change')
DEFINE FIELD(#ENBDLT) TYPE(*CHAR) LENGTH(1) LABEL('Enable Delete')
DEFINE FIELD(#TRUE) TYPE(*CHAR) LENGTH(6) LABEL('Action if True')
DEFINE FIELD(#FALSE) TYPE(*CHAR) LENGTH(6) LABEL('Action if False')
DEFINE FIELD(#MSGDET) TYPE(*CHAR) LENGTH(27) LABEL('Message Detail')
DEFINE FIELD(#MSGTXT) TYPE(*CHAR) LENGTH(80) LABEL('Message Text')
DEFINE FIELD(#USRPGM) TYPE(*CHAR) LENGTH(10) LABEL('User Program')
DEFINE FIELD(#PGMPRM) TYPE(*CHAR) LENGTH(20) LABEL('Program Parameters')
DEF_LIST NAME(#PRMWRK) FIELDS((#PGMPRM)) TYPE(*WORKING)
DEF_LIST NAME(#PRMBRW) FIELDS((#PGMPRM)) ENTRIES(10)
GROUP_BY NAME(#VALCHK) FIELDS((#LEVEL) (#FIELD) (#SEQNUM))
***** Initialize Browse list
CLR_LIST NAMED(#PRMBRW)
INZ_LIST NAMED(#PRMBRW) NUM_ENTRIES(10) WITH_MODE(*CHANGED)
***** Clear Working lists
BEGIN_LOOP
CLR_LIST NAMED(#PRMWRK)
***** Request Validation check details
REQUEST FIELDS((#VALCHK)) BROWSELIST(#PRMBRW)
***** Load key field working list
SELECTLIST NAMED(#PRMBRW)
ADD_ENTRY TO_LIST(#PRMWRK)
ENDSELECT
```

```
***** Execute Built-In Function - PUT_PROGRAM_CHECK
USE      BUILTIN(PUT_PROGRAM_CHECK) WITH_ARGS(#LEVEL #FI
***** Put "call user program" validation successful
IF      COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Put "call user program" validation check(s) was succ
***** Put "call user program" failed
ELSE
IF      COND('#RETCOD *EQ "ER"')
MESSAGE MSGTXT('Put "call user program" validation check(s) failed')
ENDIF
ENDIF
END_LOOP
```

9.193 PUT_RANGE_CHECK

⇒ **Note:** Built-In Function Rules.

Creates/amends a "range of values" DICTIONARY or FILE level validation check into the data dictionary or file definition of the nominated field.

When adding a FILE level validation check to a field, the file involved must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

All argument values passed to this Built-In Function are validated exactly as if they had been entered through the online validation check definition screen panels.

Normal authority and task tracking rules apply to the use of this Built-In Function.

For more information refer to *Field Rules and Triggers* in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	Validation performed by Visual LANSA is not as rigorous as that performed by LANSA for i.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation check. D = Dictionary level	1	1		

			F = File level				
2	A	Req	Name of field in dictionary to which validation rule is to be applied.	1	10		
3	N	Req	Sequence number of check.	1	3	0	0
4	A	Req	Description of check.	1	30		
5	A	Req	Enable check for ADD. Y = Check performed on ADD U = Check performed on ADDUSE N = Check not performed on ADD	1	1		
6	A	Req	Enable check for CHANGE. Y = Check performed on CHG U = Check performed on CHGUSE N = Check not performed on CHG	1	1		
7	A	Req	Enable check for DELETE. Y = Enable check. N = Do not enable check.	1	1		
8	A	Req	Action if check is true. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
9	A	Req	Action if check is false. NEXT = Perform next check ERROR = Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
10	A	Req	Message file details Details of error message to be issued from a message file. Message file details should be formatted as	27	27		

		Opt		Len	Len	Dec	Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

Example

A user wants to put a "range of values" validation check for a specific field, without going through the LANSAs options provided on the "Field Control Menu" that enables the user to put a "range of values" validation check.

```

***** Define arguments and lists
DEFINE FIELD(#LEVEL) TYPE(*CHAR) LENGTH(1) LABEL('Level')
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(10) LABEL('Field')
DEFINE FIELD(#SEQNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0) LABEL('Seq Num')
DEFINE FIELD(#DESCR) TYPE(*CHAR) LENGTH(30) LABEL('Description')
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2) LABEL('Return Code')
DEFINE FIELD(#ENBADD) TYPE(*CHAR) LENGTH(1) LABEL('Enable Add')
DEFINE FIELD(#ENBCHG) TYPE(*CHAR) LENGTH(1) LABEL('Enable Change')
DEFINE FIELD(#ENBDLT) TYPE(*CHAR) LENGTH(1) LABEL('Enable Delete')
DEFINE FIELD(#TRUE) TYPE(*CHAR) LENGTH(6) LABEL('Action if True')
DEFINE FIELD(#FALSE) TYPE(*CHAR) LENGTH(6) LABEL('Action if False')
DEFINE FIELD(#MSGDET) TYPE(*CHAR) LENGTH(27) LABEL('Message Detail')
DEFINE FIELD(#MSGTXT) TYPE(*CHAR) LENGTH(80) LABEL('Message Text')
DEFINE FIELD(#FRMRNG) TYPE(*CHAR) LENGTH(20) LABEL('From Range')
DEFINE FIELD(#TORNG) TYPE(*CHAR) LENGTH(20) LABEL('To Range')
DEF_LIST NAME(#FRMWRK) FIELDS((#FRMRNG)) TYPE(*WORKING)
DEF_LIST NAME(#TOWRK) FIELDS((#TORNG)) TYPE(*WORKING)
DEF_LIST NAME(#RNGBRW) FIELDS((#FRMRNG) (#TORNG)) ENTRY
GROUP_BY NAME(#VALCHK) FIELDS((#LEVEL) (#FIELD) (#SEQNUM))
***** Initialize Browse list
CLR_LIST NAMED(#RNGBRW)

```

```
INZ_LIST NAMED(#RNGBRW) NUM_ENTRYS(20) WITH_MODE(*CH/
***** Clear Working lists
BEGIN_LOOP
CLR_LIST NAMED(#FRMWRK)
CLR_LIST NAMED(#TOWRK)
***** Request Validation check details
REQUEST FIELDS((#VALCHK)) BROWSELIST(#RNGBRW)
***** Load From and To range value working lists
SELECTLIST NAMED(#RNGBRW)
ADD_ENTRY TO_LIST(#FRMWRK)
ADD_ENTRY TO_LIST(#TOWRK)
ENDSELECT
***** Execute Built-In Function - PUT_RANGE_CHECK
USE BUILTIN(PUT_RANGE_CHECK) WITH_ARGS(#LEVEL #FIELD
***** Put "range of values" validation check was successful
IF COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Put "range of values" validation check(s) was success
***** Put "range of values" failed
ELSE
IF COND('#RETCOD *EQ "ER"')
MESSAGE MSGTXT('Put "range of values" validation check(s) failed')
ENDIF
ENDIF
END_LOOP
```

9.194 PUT_REGISTRY_VALUE

⇒ **Note:** Built-In Function Rules.

Processing

Adds/updates the value for the specified Registry Key.

When the length of an Argument is stated as being greater than 50, this is only true for Fields. Literal values are restricted to a maximum length of 50. This is especially true for the first four arguments in this BIF. All these arguments are limited to a length of 50 unless a field is used.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len
1	A	R	Registry Root For example: HKEY_CLASSES_ROOT,HKEY_LOCAL_MACHINE	1	256
2	u	R	Registry Path For example: WinZip\shell\open\command	1	256
3	u	R	Registry Key Name	1	256
4	X	R	Registry Key Value: Refer to Registry Key Value Note for details.	1	Unli

5	A	O	Value Type: S – String B – Binary D – DWORD X – Expanded String which can contain environment variables. Refer to Supported & Default Key Types for further information. Default Value – S	1	1
6	A	O	Create key: Y – If registry does not exist, create one. N – No key to be created. Default – N.	1	1
7	N	O	Registry Hive to use: 32 or 64 Any other value will use the default for the application. That is, a 32 bit application will write to wow6432 while a 64 bit application will write to wow6464. This argument is ignored on a 32 bit operating system.	1	4

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Return Code OK – Key added / updated ER – Key could not be added / updated	2	2		

Registry Key Value Note

If the S type is used to save a field value (except Binary or VarBinary), the string representation of the field will be saved in the Registry.

If the B type is used to save an Alpha, Char or String field, it is presumed that the field value is a hexadecimal string and it will be converted into its actual value. Any byte in the Alpha, Char or String field must be a 1,2,3,4,5,6,7,8,9,A,B,C,D,E or F character. If a non Hexadecimal character is encountered, it will be replaced with '0' (zero).

If the D type is used to save an Alpha, Char or String field value, it is presumed that the field value is a decimal string representation of a value and it will be converted into that actual value.

The [Supported & Default Key Types](#) table shows how the key type is defined by default and also the supported key types for each field type. The default is applied when the 5th argument is not specified. It is strongly recommended that you use the default.

If a non-supported key type is used, a fatal error with the message: "Using non-supported key type" will occur.

Supported & Default Key Types

Field Type	Key Type by Default	Supported Key Type
Alpha, Char and String	S	S, B, D, X
CLOB, BLOB, Time, Date, DateTime, Packed, Signed, Float Boolean and Integer	S	S, X
Less than 8 bytes Integer	S	S, D, X
Binary VarBinary.	B	B

9.195 PUT_SYSTEM_VARIABLE

⇒ **Note:** Built-In Function Rules.

Creates / amends a system variable. If the system variable name specified does not already exist the system variable is added, if it does exist the system variable definition is updated with the new details.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	System variable name A System variable name must begin with '*'.	5	20		
2	A	Req	Description	1	40		
3	A	Req	STATIC or DYNAMIC	6	7		
4	A	Req	Data type (ALPHA, NUMBER)	5	6		
5	N	Req	Length / Total digits	3	3	0	0
6	N	Req	Decimal positions	1	1	0	0
7	A	Req	Evaluation program	1	10		
8	A	Req	Ev. program type (FUN, 3GL)	3	3		
9	A	Req	Initial public access (ALL, NORMAL or	3	6		

			NONE)				
--	--	--	-------	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code (OK, ER)	2	2		

Example

A small program to allow the creation / amendment of system variables. The user is requested to fill in the system variable details and a message based on the return code notifies if the operation was successful.

```

GROUP_BY  NAME(#SYSVAR) FIELDS(#SYSNAM #SYSDES #SYSSOI
***** Set some defaults
CHANGE   #SYSNAM *NULL
CHANGE   #SYSDES 'NULL VALUE'
CHANGE   #SYSSOD 'DYNAMIC'
CHANGE   #SYSTYP 'ALPHA '
CHANGE   #SYSLEN 1
CHANGE   #SYSDEC 0
CHANGE   #SYSPGM 'SVPGM'
CHANGE   #PGMTYP 'FUN'
CHANGE   #ACCESS 'NORMAL'
***** Request System variable details
REQUEST  FIELDS(#SYSVAR)
*****
USE      BUILTIN(PUT_SYSTEM_VARIABLE) WITH_ARGS(#SYSNAM
***** Inform user of success / failure
IF      '#RETCOD *EQ OK'
MESSAGE MSGF(SYSMSG) MSGID(SYS0023) MSGDTA(#SYSNAM)
ELSE
MESSAGE MSGF(SYSMSG) MSGID(SYS0024) MSGDTA(#SYSNAM)
* < ----- Handle any errors ----- >

```

ENDIF

9.196 PUT_TRIGGER

⇒ **Note:** Built-In Function Rules.

Creates/amends a DICTIONARY or FILE level trigger into the data dictionary or file definition of the nominated field.

When adding a FILE level trigger to a field, the file involved, must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

All argument values passed to this Built-In Function are validated exactly as if they had been entered through the online validation check definition screen panels.

Normal authority and task tracking rules apply to the use of this Built-In Function.

For more information refer to [Field Rules/Triggers](#) in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation check. D = Dictionary level F = File level	1	1		
2	A	Req	Name of field in dictionary to which trigger	1	10		

			rule is to be applied.				
3	N	Req	Sequence number of trigger.	1	3	0	0
4	A	Req	Description of trigger.	1	30		
5	A	Req	Name of trigger function.	1	7		
6	L	Req	<p>Working list of trigger points.</p> <p>The calling RDML function must provide a working list with an aggregate entry length of exactly 5 bytes and at most 6 trigger point value entries may be specified.</p> <p>Each trigger point is associated with a "before" and an "after" entry. At least one trigger point must have one of these set to "Y".</p> <p>The trigger point must be specified in 3 characters as one of:</p> <p>OPN for Open CLS for Close RED for Read INS for Insert UPD for Update DLT for Delete</p> <p>Each list entry sent should be formatted as follows:</p> <p>From - To Description 1 - 3 Trigger position 4 - 4 Trigger before 5 - 5 Trigger after</p>	5	5		
7	L	Req	<p>Working list of trigger conditions.</p> <p>The calling RDML function may provide a working list with an aggregate entry length of exactly 36 bytes and at most 20 trigger conditions entries may be specified. Each list entry sent should be formatted as follows:</p>	36	36		

		From - To	Description				
		1 - 3	Physical file name				
		1 - 3	AND / OR				
		4 - 13	Field name				
		14 - 16	Operation code				
		17 - 36	Value				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = field details returned ER = field not accessible In case of "ER" return code error message(s) are issued automatically.	2	2		

9.197 PUT_VALUE_CHECK

⇒ **Note:** Built-In Function Rules.

Creates/amends a "list of values" DICTIONARY or FILE level validation check into the data dictionary or file definition of the nominated field.

When adding a FILE level validation check to a field, the file involved must have been previously placed into an edit session by the START_FILE_EDIT Built-In Function.

All argument values passed to this Built-In Function are validated exactly as if they had been entered through the online validation check definition screen panels.

Normal authority and task tracking rules apply to the use of this Built-In Function.

For more information refer to [Field Rules/Triggers](#) in the *LANSA for i User Guide*.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	Visual LANSA validation is not as rigorous as that performed by LANSA for i.
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Level of validation check. D = Dictionary level	1	1		

			F = File level				
2	A	Req	Name of field in dictionary to which validation rule is to be applied.	1	10		
3	N	Req	Sequence number of check.	1	3	0	0
4	A	Req	Description of check.	1	30		
5	A	Req	Enable check for ADD. Y = Check performed on ADD U = Check performed on ADDUSE N = Check not performed on ADD	1	1		
6	A	Req	Enable check for CHANGE. Y = Check performed on CHG U = Check performed on CHGUSE N = Check not performed on CHG	1	1		
7	A	Req	Enable check for DELETE. Y = Enable check. N = Do not enable check.	1	1		
8	A	Req	Action if check is true. NEXT Perform next check ERROR Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
9	A	Req	Action if check is false. NEXT Perform next check ERROR Issue fatal error ACCEPT = Accept value and do no more checking.	4	6		
10	A	Req	Message file details Details of error message to be issued from a message file. Message file details should be formatted as follows:	27	27		

			From - To Description 1 - 7 Error Message Number 8 - 17 Message File Name 18 - 27 Message File Library. If message text is used, pass this argument as blanks.				
11	A	Req	Message text.	1	80		
12	L	Req	Working list to contain list values. The calling RDML function must provide a working list with an aggregate entry length of exactly 20 bytes and at most 50 list value entries may be specified. Each list entry sent should be formatted as follows: Bytes 1-20: List value	20	20		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = validation check defined ER = fatal error detected In case of "ER" return code error message(s) are issued automatically. When a file edit session is involved it is ended automatically without commitment.	2	2		

Example

A user wants to put a "list of values" validation check for a specific field, without going through the LANSAs options provided on the "Field Control Menu" that enables the user to put a "list of values" validation check.

```

***** Define arguments and lists
DEFINE FIELD(#LEVEL) TYPE(*CHAR) LENGTH(1) LABEL('Level')
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(10) LABEL('Field')
DEFINE FIELD(#SEQNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0) LABEL('Seq Num')
DEFINE FIELD(#DESCR) TYPE(*CHAR) LENGTH(30) LABEL('Description')
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2) LABEL('Return Code')
DEFINE FIELD(#ENBADD) TYPE(*CHAR) LENGTH(1) LABEL('Enable Add')
DEFINE FIELD(#ENBCHG) TYPE(*CHAR) LENGTH(1) LABEL('Enable Change')
DEFINE FIELD(#ENBDLT) TYPE(*CHAR) LENGTH(1) LABEL('Enable Delete')
DEFINE FIELD(#TRUE) TYPE(*CHAR) LENGTH(6) LABEL('Action if True')
DEFINE FIELD(#FALSE) TYPE(*CHAR) LENGTH(6) LABEL('Action if False')
DEFINE FIELD(#MSGDET) TYPE(*CHAR) LENGTH(27) LABEL('Message Detail')
DEFINE FIELD(#MSGTXT) TYPE(*CHAR) LENGTH(80) LABEL('Message Text')
DEFINE FIELD(#LSTVAL) TYPE(*CHAR) LENGTH(20) LABEL('List Value')
DEF_LIST NAME(#VALWRK) FIELDS((#LSTVAL)) TYPE(*WORKING)
DEF_LIST NAME(#VALBRW) FIELDS((#LSTVAL)) ENTRIES(50)
GROUP_BY NAME(#VALCHK) FIELDS((#LEVEL) (#FIELD) (#SEQNUM))
***** Initialize Browse list
CLR_LIST NAMED(#VALBRW)
INZ_LIST NAMED(#VALBRW) NUM_ENTRIES(50) WITH_MODE(*CHAR)
***** Clear Working list
BEGIN_LOOP
CLR_LIST NAMED(#VALWRK)
***** Request Validation check details
REQUEST FIELDS((#VALCHK)) BROWSELIST(#VALBRW)
***** Load list of values working list
SELECTLIST NAMED(#VALBRW)
ADD_ENTRY TO_LIST(#VALWRK)
ENDSELECT
***** Execute Built-In Function - PUT_VALUE_CHECK
USE BUILTIN(PUT_VALUE_CHECK) WITH_ARGS(#LEVEL #FIELD)
***** Put "list of values" validation check was successful
IF COND('#RETCOD *EQ "OK"')
MESSAGE MSGTXT('Put "list of values" validation check(s) was successful')
***** Put "list of values" failed
ELSE
IF COND('#RETCOD *EQ "ER"')
MESSAGE MSGTXT('Put "list of values" validation check(s) failed')
ENDIF

```

```
ENDIF  
END_LOOP
```

9.198 PUT_WEB_COMPONENT

⇒ **Note:** Built-In Function Rules.

Programmatically (re)build an HTML Web Component (refer to the LANSAs for the Web Guide for more information about Web Components).

Use this Built-In Function when you want to generate handcrafted web components at execution time. Note that there is no validation of the component page text; it is your responsibility to verify that your web components work as expected.

This Built-In Function only supports page, script, text, and visual components. Any existing component will be replaced unless the existing component is of an unsupported type, in which case an error will occur.

If a component already exists the function will fail if the existing component is not of the same type as the one used in this function.

For use with

LANSAs for i	YES	Only available for RDMLX.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Web Component	1	20		
2	A	Req	Mode. Possible values are: I: Input O: Output N: Not applicable.	1	1		
3	A	Req	Type. Possible values are: P: Page	1	1		

			S: Script T: Text V: Visual.				
4	A	Req	Description	1	40		
5	L	Req	Working list containing component page text. For a Text component, the working list should only have one entry.	255	255		
6	A	Opt	Page Defaults to same as component. Generally, you should leave as default.	1	20		
7	A	Opt	Language Default is current language For non-multilingual partitions this should be 'NAT' *DFT - partition default language.	4	4		
8	A	Opt	Roll sets? (Y/N) Default is N.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Possible values are: OK: Build completed normally. ER: Build encountered an error.	1	2		

9.199 RANDOM_NUM_GENERATOR

⇒ **Note:** Built-In Function Rules.

Returns a random number between 0 and 1.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Opt	Stream index	1	2	0	0
2	N	Opt	Seed	1	10	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Random number between 0 and 1.	1	63	1	63

Technical Notes

- By default, the Built-In-Function RANDOM_NUM_GENERATOR always returns a random number between 0 and 1. For advanced users, it also provides 100 independent streams of random numbers, where drawing a random number from one stream does not affect the sequence of random

numbers drawn from a different stream. Each stream is referenced using an optional "stream index" in the range [0,99], inclusive. If no stream index is supplied, then stream 0 is used by default.

- By default, the Built-In-Function RANDOM_NUM_GENERATOR seeds (i.e. initializes) each stream from the current system time the first time the stream is referenced. For advanced users, it also accepts an optional seed value which is used to seed (i.e. initialize) a stream. This is useful if you wish to repeat the same sequence of random numbers many times.

Examples

Get the next random number from the default stream (0):

```
Use Builtin(RANDOM_NUM_GENERATOR) To_Get(#x_dec)
```

Get the next random number from stream 23:

```
Use Builtin(RANDOM_NUM_GENERATOR) With_Args(23)
To_Get(#x_dec)
```

Seed stream 42 with the value 12345. n.b. This seeds the stream and so will always return the same random number value:

```
Use Builtin(RANDOM_NUM_GENERATOR) With_Args(42 12345)
To_Get(#x_dec)
```

Seed the default stream (0) with the value 56789.

Note: This seeds the stream and so will always return the same value:

```
Use Builtin(RANDOM_NUM_GENERATOR) With_Args(*DEFAULT
56789) To_Get(#x_dec)
```

9.200 RCV_FROM_DATA_QUEUE

⇒ **Note:** Built-In Function Rules.

Receives one or more working list's entries from an IBM i or Windows emulated data queue. For more information about data queues refer to the appropriate IBM manuals.

Note: Only use this Built-In Function in applications that are to fully execute under the control of the IBM i or a Windows operating system.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	A literal or variable that specifies or contains the name of the data queue. This name must conform to IBM i object naming conventions. This is not checked by the Built-In Function.	1	10		
2	N	Req	A literal or variable that specifies or contains the byte length of one complete entry of the working list specified in return value 1.	1	5	0	0
3	N	Req	A literal or variable that specifies or contains the length of time (in seconds) that the Built-In Function should wait for data to arrive on	1	5	0	0

		<p>the data queue.</p> <ul style="list-style-type: none"> - A negative value indicates an unlimited wait. - A zero value indicates no wait is required. - A Positive value is the number of seconds. <p>Refer to the appropriate IBM supplied manual for more details of this argument. Refer to IBM supplied program QRCVDTAQ which is what is actually used by this Built-In Function.</p>				
--	--	---	--	--	--	--

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	The name of the working list whose entries are to be received from the specified data queue.	1	10		

Technical Notes

- This Built-In Function may be used with IBM i or with a Windows operating system. Please refer to the Technical notes associated with the SND_TO_DATA_QUEUE Built-In Function for more information about using data queues under different operating systems.

Examples

Receive a customer number and a part number from a data queue called PICKLIST and then print their details. Assume that the entry is already known to be on the data queue:

```
DEF_LIST NAME(#PICK) FIELDS(#CUSTNO #PARTNO) TYPE(*WORK)
      (where #CUSTNO is defined in the dictionary as a signed 5,0 number an
```

```
USE    BUILTIN(RCV_FROM_DATA_QUEUE) WITH_ARGS('PICKLIST
GET_ENTRY NUMBER(1) FROM_LIST(#PICK)
EXECUTE SUBROUTINE(PRINT_PICK)
```

Sit in a permanent loop receiving customer and part number details (one by one) as they arrive. As each arrives its details should be printed:

```
DEF_LIST NAME(#PICK) FIELDS(#CUSTNO #PARTNO) TYPE(*WORK
BEGIN_LOOP
USE    BUILTIN(RCV_FROM_DATA_QUEUE) WITH_ARGS('PICKLIST
GET_ENTRY NUMBER(1) FROM_LIST(#PICK)
EXECUTE SUBROUTINE(PRINT_PICK)
END_LOOP
```

Sit in a permanent loop receiving customer and part number details (in blocks of up to 5) as they arrive. As each block arrives it should be printed:

```
DEF_LIST NAME(#PICK) FIELDS(#CUSTNO #PARTNO) TYPE(*WORK
BEGIN_LOOP
USE    BUILTIN(RCV_FROM_DATA_QUEUE) WITH_ARGS('PICKLIST
BEGIN_LOOP USING(#I) FROM(1) TO(#LISTCOUNT)
  GET_ENTRY NUMBER(#I) FROM_LIST(#PICK)
  EXECUTE SUBROUTINE(PRINT_PICK)
END_LOOP
END_LOOP
```

Note: Routines placing customer/part number pairs onto this data queue can actually place 1,2,3,4 or 5 entries and this function will work successfully. However if a function attempted to place more than 5 entries onto one data queue entry, then this application would fail because working list #PICK can contain at most 5 entries.

9.201 REBUILD_FILE

⇒ **Note:** Built-In Function Rules.

Optionally drops the existing file and its views, and creates a new file from the CTD file.

REBUILD_FILE can also be used to alter a PC OTHER File to add the two LANSAs columns X_RRNO and X_UPID.

For use with

LANSAs for i	NO
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	NO

Processing

If drop views specified, drop the table and its matching views.

If Create new file is specified, create the new table from the CTD file.

If Use Connection Parameters in CTD file is specified, alter the table in the external database specified in the CTD file.

If Prompt for user ID and password is specified, remove the existing user id and password from the connection parameters in the CTD file and allow the database manager (e.g. ODBC driver) to prompt for the user id and password. Note that some database managers prompt for much more than this, even though it is only the user id and password that is required. Some others do not need a user id and password and so the database manager will still not prompt for them (e.g. MS Access and MS SQL Server).

Arguments

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	The LANSAs File Name	1	10		
2	A	Opt	Drop File and Views	1	1		

			Values: Y = Drop N = Do not Drop Default = Y				
3	A	Opt	Create New File Values: Y = Create N = Do not Create Default = Y	1	1		
4	A	Opt	Use Connection Parameters in CTD File Values: Y = Connect using CTD File N = Use current database connection Default = N	1	1		
5	A	Opt	Prompt for user ID and password Values: Y = Prompt N = Use CTD file user id and password Default = N	1	1		
6	A	Opt	Library Name	1	10		
7	A	Opt	CTD Location Level A= All (Partition + System). P = Partition Level only. S=System Level only. Default is A.	1	1		

Return Values

--	--	--	--	--	--	--	--

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code: OK = File successfully rebuilt ER = An Error occurred rebuilding the file	2	2		

9.202 REBUILD_TABLE_INDEX

⇒ **Note:** Built-In Function Rules.

Clears and rebuilds the high speed index entries associated with file definition(s) that are flagged as high speed tables. Refer to the [Database File Attributes](#) section in the *LANSA for i User Guide* for more information about IBM i high speed tables.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	A literal or variable that specifies or contains the name of the file. The name may be special value "*ALL", a full name, or a generic name	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code. Returned values possible are: OK: Rebuild(s) all completed normally. ER: One or more rebuild(s) encountered an error.	2	2		

		NR: No files flagged as high speed tables found matching the name supplied.				
--	--	--	--	--	--	--

Examples

Rebuild the index of a file called STATES:

```
USE BUILTIN(REBUILD_TABLE_INDEX) WITH_ARGS('STATES') TO
```

Rebuild the index of all files that start with S:

```
USE BUILTIN(REBUILD_TABLE_INDEX) WITH_ARGS('S') TO_GET
```

Delete the user index area and completely rebuild it:

```
EXEC_OS400 COMMAND('DLTUSRIDX DC@TBLIDX')
```

```
USE BUILTIN(REBUILD_TABLE_INDEX) WITH_ARGS('*ALL') TC
```

9.203 RESET_@@UPID

⇒ **Note:** Built-In Function Rules.

Resets the @@UPID field to zero.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	File name	1	10		
2	A	Opt	Library name Default '*LIBL'	1	10		

Return Values

No return values.

This function is used to reset the value of the @@UPID field to zero.

Example

```
DEFINE  FIELDS(#PARTDTALIB) TYPE(*CHAR) LENGTH(10) DEFAU  
USE    BUILTIN(RESET_@@UPID) WITH_ARGS(#FILENAME #PARTI
```

9.204 RESTORE_SAVED_LIST

⇒ **Note:** Built-In Function Rules.

Restores the contents of a previously saved permanent or temporary working list.

Portability Considerations RESTORE_SAVED_LIST and SAVE_LIST should be used in the same context. That is, you should not use SAVE_LIST from within a function run on the WEB and later restore the saved list from within a function run as a Visual LANSA function.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of previously saved list to be restored.	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working List for saved list to be loaded/restored into.	1	10		

2	A	Opt	Return Code 'OK' - List retrieved 'OV' - List full 'NR' - List not found	2	2		
3	N	Opt	Length of each list entry	1	15	0	0
4	A	Opt	Type of list 'P' - Permanent list 'T' - Temporary List	1	1		

Technical Notes

- This Built-In Function is designed to restore a saved working list, or to retrieve the contents of a working list passed between jobs (e.g. interactive to batch). It is **not** designed for retrieving working list details of a program in the same job. The exchange of working list details between programs in the same job can be achieved in a much more efficient manner by using the RCV_LISTS and PASS_LISTS parameters of the FUNCTION and CALL commands.
- File DC@F80 in the LANSAs data library is used to store saved list details. This file should be considered in your backup and restore procedures.
- The target list must be the same size or bigger than the saved list or none of the information saved will be restored. This scenario will return an 'OV' return code and an empty list.
- Deleted record space in file DC@F80 is reorganized and removed during a normal LANSAs internal database reorganization. This reorganization also deletes any temporary lists that have exceeded their retention period.
- You can reorganize file DC@F80 to free deleted record space at any time by using the IBM i RGZPFM (Reorganize Physical File Member) command. Use DC@F80V1 as the sequencing logical view.
- The backup and recovery of data area DC@A08 and database file DC@F80 (and its logical views DC@F80V1 and DC@F80V2) is **your** responsibility.
- Movement of DC@F80 or saved lists between machines or between environments is your responsibility.

Example

A list has been saved. This list is a list of clients that have been selected by user to be printed on a report. The name of the list has been passed to this job by the submitting job.

```
DEF_LIST NAME(#RSTLST) FIELDS((#CLICDE) (#CLIDES)) TYPE(*W
DEF_LINE NAME(#LINE1) FIELDS((#CLICDE) (#CLIDES) (#CLIAD1) (
DEFINE FIELD(#LSTNME) TYPE(*CHAR) LENGTH(10)
***** Clear the list
CLR_LIST NAMED(#RSTLST)
***** Restore the list
USE BUILTIN(RESTORE_SAVED_LIST) WITH_ARGS(#LSTNME) TO
***** Process the list
SELECTLIST NAMED(#RSTLST)
FETCH FIELDS(#LINE1) FROM_FILE(CLIMASTER) WITH_KEY(#CL
PRINT LINE(#LINE1)
ENDSELECT
***** Close print file
ENDPRINT
***** Submit job to delete list
USE BUILTIN(DELETE_SAVED_LIST) WITH_ARGS(#LSTNME)
```

9.205 REVERSE

⇒ **Note:** Built-In Function Rules.

Reverses a text string.

Important note for DBCS: This Built-In function is DBCS unaware. To reverse a string containing DBCS characters so that the DBCS characters do not get corrupted you must use RDMLX .Reverse intrinsic

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to reverse	1	Unlimited		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Reversed string	1	Unlimited		

Examples

Example 1: Reverse a text string input from user

```
DEFINE FIELD(#INTEXT) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#OUTTEXT) TYPE(*CHAR) LENGTH(20)
```

```
REQUEST  FIELDS(#INTEXT)
USE      BUILTIN(REVERSE) WITH_ARGS(#INTEXT) TO_GET(#OUTE)
DISPLAY  FIELDS(#OUTEXT)
```

Example 2: Reverse a known text string

```
DEFINE   FIELDS(#OUTEXT) TYPE(*CHAR) LENGTH(10)
*****
USE      BUILTIN(REVERSE) WITH_ARGS("ti esreveR")
        TO_GET(#OUTEXT)
DISPLAY  FIELDS(#OUTEXT)
```

9.206 RIGHT

⇒ **Note:** Built-In Function Rules.

Right aligns argument into return string.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be right align	1	256		
2	A	Opt	Remove imbedded blanks flag (Y/N) Values: Y = remove N = do not remove Default: N	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return right aligned string	1	256		

Example

Right align and remove imbedded blanks from a requested string.

```
DEFINE FIELD(#INTEXT) TYPE(*CHAR) LENGTH(18)
DEFINE FIELD(#OUTTEXT) TYPE(*CHAR) LENGTH(18)
*****
REQUEST FIELDS(#INTEXT)
USE BUILTIN(RIGHT) WITH_ARGS(#INTEXT Y) TO_GET(#OUTTEXT)
DISPLAY FIELDS(#OUTTEXT)
```

Resulting displays would look something like this:

```
FUN01      Right Example
```

```
In text . . .  FR  E  D
```

```
CF1=Help
```

then,

```
FUN01      Right Example
```

```
Out text . . .          FRED
```

```
CF1=Help
```

9.207 ROUND

⇒ **Note:** Built-In Function Rules.

Rounds off a decimal value.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Value to be rounded	1	15	0	9

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Returned value	1	15	0	9

Note: This function will round off a numeric value. The length of the return value field will determine where the value will be rounded.

Example

To round a 3 decimal field to 2 decimal places.

```
DEFINE FIELD(#INVAL) TYPE(*DEC) LENGTH(5) DECIMALS(3)
DEFINE FIELD(#OUTVAL) TYPE(*DEC) LENGTH(4) DECIMALS(2)
```

USE BUILTIN(ROUND) WITH_ARGS(#INVAL) TO_GET(#OUTVAL)

9.208 SAVE_LIST

⇒ **Note:** Built-In Function Rules.

Permanently or temporarily saves the contents of a working list.

Portability SAVE_LIST and RESTORE_SAVED_LIST should be used in the same context. That is, you should not use SAVE_LIST from within a function run on the WEB and later restore the saved list from within a function run as a Visual LANSA function.

Considerations

For use with

LANSA for i	YES	
Visual LANSA for Windows	YES	Working list data must not contain X'FF' characters. Refer to the topics IBM i Defined Data Areas and Data Areas and Other LANSA Features in the <i>LANSA Application Design Guide</i> for information about data areas in this environment.
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Working List to be saved	1	10		
2	N	Req	Length of each list entry i.e. Total length of all fields in the entry. For packed fields allow (SIZE/2)+1. This parameter is ignored for RDMLX lists. Any value can be specified. Whatever the list	1	15	0	0

			entry length is, it will be saved correctly.				
3	A	Opt	Type of list 'P' - Permanent list 'T' - Temporary List If not specified, the list is assumed to be temporary.	1	1		
4	N	Opt	Retention period (in days) for temporarily saved lists. Only use a value in the range 1 to 90. If not specified, a default value of 7 is assumed for temporary lists.	1	2	0	0
5	A	Opt	Name that list is to be saved with. If this argument is not specified, or passed as blank a unique list name will be automatically assigned.	10	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Returned name of saved list	10	10		

Technical Notes

- This Built-In Function is designed to permanently save a working list, or to pass the contents of a working list between jobs (e.g. interactive to batch). It is **not** designed to exchange working lists details between programs in the same job. The exchange of working list details between programs in the same job can be achieved in a much more efficient manner by using the RCV_LISTS and PASS_LISTS parameters of the FUNCTION and CALL commands.

- Automatically assigned list names are assigned from data area DC@A08 in the LANSAs data library. In system restore situations the value in this data area should be restored as well, and then verified as being a larger number than the highest valued record found in file DC@F80.
- The automatic list name assignment procedure locks and retrieves data area DC@A08, then increments its value by 1, then returns the new value to the data area and finally unlocks it. The data area contains a 9 digit number, and the final 10 character list name is created by prefixing the assigned 9 digits with the unique prefix of the current partition.
- The automatic list name assignment logic prevents the same name from ever being assigned to more than one executing IBM i task. However, this may not be the case when you are using manually assigned list names. For instance if two IBM i tasks attempt to save a list with manually assigned name 'TESTLIST', unpredictable results may occur.
- File DC@F80 in the LANSAs data library is used to store saved list details. This file should be considered in your backup and restore procedures.
- Deleted record space in file DC@F80 is reorganized and removed during a normal LANSAs internal database reorganization. This reorganization also deletes any temporary lists that have exceeded their retention period.
- You can reorganize file DC@F80 to free deleted record space at any time by using the IBM i RGZPFM (Reorganize Physical File Member) command. Use DC@F80V1 as the sequencing logical view.
- It is good practice to specifically delete temporary lists use the DELETE_SAVED_LIST Built-In Function, rather than waiting for them to exceed their retention period and thus be automatically deleted during the next internal reorg.
- The backup and recovery of data area DC@A08 and database file DC@F80 (and its logical views DC@F80V1 and DC@F80V2) is **your** responsibility.
- Movement of DC@F80 or saved lists between machines or between environments is your responsibility.

Example

A list is displayed to the user. The user can select entries from the list to be output to a printer. By saving a list of selected entries this can later be restored by a print job to create the output.

```
DEF_LIST  NAME(#CLIENTS) FIELDS((#SELECTOR *SEL) #CLICDE #(
DEF_LIST  NAME(#SAVLST) FIELDS(#CLICDE #CLIDES) TYPE(*WOR
```

```
DEFINE FIELD(#LSTNME) TYPE(*CHAR) LENGTH(10)
***** Clear the list
CLR_LIST NAMED(#CLIENTS)
CLR_LIST NAMED(#SAVLST)
***** Build the browselist
SELECT FIELDS(#CLIENTS) FROM_FILE(CLIMASTER)
ADD_ENTRY TO_LIST(#CLIENTS)
ENDSELECT
***** Allow user to select clients for print
DISPLAY BROWSELIST(#CLIENTS)
SELECTLIST NAMED(#CLIENTS) GET_ENTRY(*SELECT)
ADD_ENTRY TO_LIST(#SAVLST)
ENDSELECT
***** Save the list
USE BUILTIN(SAVE_LIST) WITH_ARGS(#SAVLST 50 T 10) TO_GET
***** Submit job to print client information
SUBMIT PROCESS(PRINTS) FUNCTION(CLIENTS) EXCHANGE(#LS
```

9.209 SCANSTRING

⇒ **Note:** Built-In Function Rules.

Scans a string for the first occurrence of a pattern.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

This table shows the arguments used in this function.

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to be scanned	1	Unlimited		
2	A	Req	Pattern to be located. Enter in upper and/or lower case as required. This parameter is ignored for RDMLX Lists. Any value can be specified. Whatever the list entry length is, it will be saved correctly.	1	Unlimited		
3	N	Opt	Position to start scan Range: 1 - maximum possible length of the string to be scanned. Default: 1	1	11	0	0
4	A	Opt	Compare in uppercase? 1 = No, compare all cases 0 = Yes, compare upper case. Default: 1	1	1		

			See Note .				
5	A	Opt	Trim trailing pattern blanks? 1 = Yes 0 = No Default: 1 -trim trailing blanks from wild card pattern.	1	1		
6	A	Opt	Wild card in scan pattern Values: Blank = no wild card activated. Non-blank = wild card activated. Default: no wild card. Do not use a blank as the left most character of the pattern. An error will result if you do this.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Position of first occurrence in string or error code. Any one of these values may be returned: +p = Occurrence of pattern found at position "p" 0 = Pattern not found in string to be scanned. -1 = Error. Pattern is longer than string. -2 = Error. Pattern length is less than 1. -3 = Error. First char in pattern is wild.	1	3	0	0

		-4 = Error. Pattern is blank and trim requested.				
		-5 = Error. Starting position is invalid .				

Note

The result is converted to upper case for string comparison. Following is an example of some results.

String to search for	Upper Case parameter?	Result	Found?
ABC	1 (No)	123abc4	Yes
		123ABC4	Yes
ABC	0 (Yes)	123abc4	No
		123ABC4	Yes
abc	1 (No)	123abc4	No
		123ABC4	No
abc	0 (Yes)	123abc4	Yes
		123ABC4	No

Example

This code searches for the string "where" in the text "find where it exists".

```
Function Options(*DIRECT)
Define Field(#PATTERN) Reffld(#SKILCODE) Label('Find Pattern') Defau
Define Field(#STARTPOS) Reffld(#STD_IDNOS) Label('Start Pos') Defau
Override Field(#STD_FLAG) Label('Case (1/0)?') Default("1")
Define Field(#TRIM) Reffld(#STD_FLAG) Label('Trim (1/0)') Default("1"
Define Field(#WILD) Reffld(#STD_FLAG) Label('WildCard?') Default("")
Override Field(#STD_IDNOS) Label('Occurs at Pos.') Edit_Code(L)
Change Field(#STD_TEXTS) To("Find where it exists")
```

```
Begin_Loop
Request  Fields((#STD_TEXTS *LOWER) #PATTERN #STARTPOS #STD
Use      Builtin(SCANSTRING) With_Args(#STD_TEXTS #PATTERN #ST
End_Loop
```

9.210 SELECT_IN_SPACE

⇒ **Note:** Built-In Function Rules.

Selects the first row of cells that matches the key values supplied and returns the cell values into the specified fields.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ A/N	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A		R	Space Name	1	256		
2- 20	w		O	Fields that specify the key values to be used to locate the first cell row required.	1	Unlimited	0	Unlimited

Return Values

No	Type	Req/ A/N	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A		O	Standard Return Code "OK" = A cell row was found and the cell values have been returned. "NR" = No cell row could be found with a key matching the key	2	2		

			values supplied. "ER" = Select attempt failed. Messages issued will indicate more about the cause of the failure.				
2-20	w	O	Fields to receive the values of the cells in the selected cell row.	1	Unlimited	0	Unlimited

Technical Notes

The return fields must be specified in the same order as the cells in the space were defined. Cells are matched by the order of their specification in return values 2 -> 20. The names of the fields used have no bearing whatsoever on the cell mapping logic.

You can specify less key values than are defined in the space. The first matching cell row will be returned. This means that partial key operations can be performed.

If you specify more key values than are defined as key cells for the space then the additional values will be ignored and have no effect on the outcome of the search.

If you specify less return field values than there are cells in the space then the non-specified cells are not mapped back into the fields.

If you specify more return field values than there are cells in the space then the additional field values are ignored and are not changed by the search operation.

If a key value longer than 256 bytes is specified, a fatal error will occur.

Example

This example is selecting the first record with a matching surname.

```

Define Field(#SpaceRC) Type(*Char) Length(2)
Def_Cond Name(*Okay) Cond('#SpaceRC = OK')
*
Use Builtin(Select_in_Space) With_Args(TEST #SURNAME)
To_Get(#SpaceRC #SURNAME #GIVENAME #EMPNO)
Dowhile (*Okay)
Add_Entry To_List(#Emplist)

```

```
Use Builtin(SelectNext_in_Space) With_Args(TEST #SURNAME)
To_Get(#SpaceRC #SURNAME #GIVENAME #EMPNO)
Endwhile
```

9.211 SELECTNEXT_IN_SPACE

Selects the next row of cells that matches the key values supplied and returns the cell values into the specified fields.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name	1	256		
2-20	w	O	Fields that specify the key values to be used to locate the next cell row required..	1	Unlimited	0	Unlimited

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "OK" = A cell row was found and the cell values have been returned. "NR" = No cell row could be found with a key matching the key values supplied. "ER" = Select attempt failed. Messages issued will indicate more	2	2		

			about the cause of the failure.				
2-20	w	O	Fields to receive the values of the cells in the selected cell row.	1	Unlimited	0	Unlimited

Technical Notes

The return fields must be specified in the same order as the cells in the space were defined. Cells are matched by the order of their specification in return values 2 -> 20. The names of the fields used have no bearing whatsoever on the cell mapping logic.

You can specify less key values than are defined in the space. The first matching cell row will be returned. This means that partial key operations can be performed.

If you specify more key values than are defined as key cells for the space then the additional values will be ignored and have no effect on the outcome of the search.

If you specify less return field values than there are cells in the space then the non-specified cells are not mapped back into the fields.

If you specify more return field values than there are cells in the space then the additional field values are ignored and are not changed by the search operation.

If a key value longer than 256 bytes is specified, a fatal error will occur.

Using a `SELECTNEXT_IN_SPACE` operation that is not immediately preceded by a `SELECT_IN_SPACE` or a `SELECTNEXT_IN_SPACE` operation may produce unpredictable results and/or application failure.

2	A	Opt	<p>Pull down choice (PD\$OPT) value of pull down choice to be changed.</p> <p>Allowable values are: CUR- Current choice</p> <p>ALL- All choices</p> <p>other - PD\$OPT value as it is specified in the action bar control table.</p> <p>Default value is ALL.</p>	1	3		
3	A	Opt	<p>Set pull down choice to available or unavailable. Allowable values are:</p> <p>Y - Set as available.</p> <p>N - Set as unavailable.</p> <p>Default value is Y.</p>	1	1		

Return Values

No return values.

9.213 SET_AUTHORITY

⇒ **Note:** Built-In Function Rules.

Sets the authority of a user to a LANSAs object. The caller of this Built-In Function must have management rights (MD) to the LANSAs object.

Special Notes: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML applications.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted

Changes to a user's access rights to LANSAs objects may not take effect until the next time the user starts to use LANSAs. If the user is currently using LANSAs they should exit from LANSAs and then re-invoke LANSAs to ensure that the changed object access rights take effect.

This condition also applies to the caller of SET_AUTHORITY changing their own authorities to objects.

Changes to authority will have no impact if security checking is disabled for the type of object (process, function, or file). To determine your security settings on IBM i refer to [Execution and Security Settings](#) in the Review System Settings facility described in the *LANSAs for i User Guide*. To determine your security settings on Windows, or applications deployed to Linux, refer to the x_defppp.h file that you compiled your processes and forms with. You can find out more about this file in [The X_DEFppp.H Definition Header File](#).

For use with

LANSAs for i	YES	Not available for RDMLX.
Visual LANSAs for Windows	YES*	Object Types P# (Partition) and AT (Application Template) have no meaning to Visual LANSAs. If passed to this Built-In Function an error is returned.
Visual LANSAs for Linux	YES	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object name.	1	10		
2	A	Req	Object extension.	1	10		
3	A	Req	Object type. Valid types are: AT - Application template* DF - Field FD - File PD - Process PF - Function P# - Partition* SV - System variable MT - Multilingual variable. *Do not use with Visual LANSA.	2	2		
4	A	Req	User name.	1	10		
5	A	Req	Access rights This is a string of 2 character codes representing the different access rights that the user is to have. The individual access rights are: UD - Use Definition MD - Manage Definition DD - Existence of Definition DS - Data - Display AD - Data - Add CH - Data - Change DL - Data - Delete If the entire string is blank then the user is to have their access rights to the object revoked.	1	20		

		If the string has the special value '*DELETE' then the user's authority is to be deleted, thus causing their rights to revert back to their associated group profile or *PUBLIC.				
--	--	--	--	--	--	--

Dependencies

Object Type	Object Name	Object Extension
AT	template name	*blank
DF	field name	*blank
FD	file name	*blank, *LIBL, library name
PD	process name	*blank
PF	process name	function name
P#	partition name	*blank
SV	positions 1-10 of system variable name	positions 11-20 of system variable name
MT	positions 1-10 of multilingual variable name	positions 11-20 of multilingual variable name

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = The authority of the user to the LANSAs object has been set. ER = Error occurred in setting the authority of the user to the object.	2	2		

9.214 SET_DD_ATTRIBUTES

⇒ **Note:** Built-In Function Rules.

Allows the characteristics of a field that is to be visualized as a drop-down to be controlled.

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	The name of the drop down. It must begin with "DD" and be in the format DDxx. This name corresponds to the name used in the field's input or output attributes and the name used in the ADD_DD_VALUES Built-In Function.	4	4		
2	A	Req	The style to be used when visualizing the drop down field. Allowable values are: A - The default single line non-editable drop down. B - A single line editable drop down. C - A multiple line editable drop down. Under Windows this style is visualized as a combo-box. D - A multiple line non-editable drop down. This style is visualized as a list box.	1	1		

			Invalid values in this argument are ignored and they are converted to style A.				
3	N	Opt	<p>The total number of panels "lines" that the drop down control is to occupy on the panel or window.</p> <p>For styles A and B this value indicates the number of lines that the drop down should occupy when the user has dropped it down.</p> <p>For styles C and D this value indicates the number of lines that the drop down control should permanently occupy on the panel.</p> <p>This argument must be a valid number in the range 1 to 20. Values outside this range are ignored and the default value is used.</p> <p>The value is specified in panel "lines", where a panel is up to 24 lines long. The actual number of items that can be visualized in the drop down depends upon the height of the font being used.</p> <p>The default length is the least of</p> <p>a) no. of items in the List;</p> <p>b) remaining lines on the Panel.</p>	1	7	0	00

Return Values

No return values.

General Technical Notes

- Once a drop down's attributes are set they remain in effect for the duration of the current job / process. Subsequent use of this Built-In Function for the same drop down name will effectively replace (ie: overwrite) its current attribute values.
- In all styles, using a length of 1 is fairly nonsensical.
- If the length value is set so that the drop down extends outside of the current

panel or pop-up window then unpredictable results and/or application failure may result.

- For styles C and D, setting the length value so that the drop down overlaps any other object in the panel or window may cause unpredictable results and/or application failure.
- You cannot put duplicate data into a drop down. Duplicated items in a drop down are treated as the same item. You must ensure that your application does NOT put duplicated data into a drop down.
- There are finite limits to the total storage occupied by all the items in a drop down. Typically this is 32K, but you should never be anywhere near this limit because of usability limitations.
- There are finite limits to the usability of drop downs that are met before storage limits are exceeded. Typically drop downs contain up to around 100 items. Drop downs containing thousands of items are not advisable and should not be used.

9.215 SET_FILE_ATTRIBUTE

⇒ **Note:** Built-In Function Rules.

Sets a file's database attributes.

An edit session must be commenced by using the START_FILE_EDIT Built-In Function prior to using this Built-In Function.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Attribute to set	1	256		
			Valid attributes . . . I/O module: 'IOMODULE=YES' 'IOMODULE=NO ' IBM i High Speed Table: 'OS400HST=YES' 'OS400HST=NO '				

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code (OK, ER) In case of "ER" return code error message(s) are issued automatically.	2	2		

Note: Currently this Built-In Function can only be used to determine whether or not a file will have an I/O module.

Example

A LANSAs function to emulate the 'File definition Menu' has been written. When a certain option is taken the user can decide to set a file attribute. IE Do you want an I/O module (Yes/No) ?

```

***** Define arguments and lists
DEFINE FIELD(#FILNAM) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#LIBNAM) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#YESNO) TYPE(*CHAR) LENGTH(32) LABEL('I/O Module')
DEFINE FIELD(#RETCOD) TYPE(*CHAR) LENGTH(2)
BEGIN_LOOP
***** Request File and library name and I/O module attribute
REQUEST FIELDS(#FILNAM #LIBNAM #YESNO)
*****
USE BUILTIN(START_FILE_EDIT) WITH_ARGS(#FILNAM #LIBNAM)
IF COND('#YESNO *EQ YES')
USE BUILTIN(SET_FILE_ATTRIBUTE) WITH_ARGS('IOMODULE=')
ELSE
USE BUILTIN(SET_FILE_ATTRIBUTE) WITH_ARGS('IOMODULE=')
ENDIF
USE BUILTIN(END_FILE_EDIT) WITH_ARGS('Y') TO_GET(#RETCOD)
***** Submit job to make file operational
USE BUILTIN(MAKE_FILE_OPERATIONL) WITH_ARGS(#FILNAM)
TO_GET(#RETCOD)
*****
END_LOOP

```

9.216 SET_FOR_HEAVY_USAGE

⇒ **Note:** Built-In Function Rules.

Sets a function for heavy usage mode

For use with

LANSA for i YES

Visual LANSA for Windows NO

Visual LANSA for Linux NO

Arguments

No Argument Values.

Return Values

No Return Values.

Technical Notes

- This Built-In Function allows a function to dynamically change from heavy usage to light usage (and vice-versa). For example:

```
if (*jobmode = B)
```

```
    use SET_FOR_HEAVY_USAGE
```

```
else
```

```
    use SET_FOR_LIGHT_USAGE
```

```
endif
```

- At every entry or (re)entry the heavy/light usage option is set by the calling parent process, or adopted from a *DIRECT caller, so at every invocation you should positively (re)set the usage option.

- This Built-In Function executes very quickly and imposes little overhead.
- Changing a "light usage" function to have a "heavy usage" capability may be more complex than simply adding a use of the SET_FOR_HEAVY_USAGE Built-In Function. Since a heavy usage function's variables and lists retain their values between invocations, more complex entry and/or exit logic may be required.

9.217 SET_FOR_LIGHT_USAGE

⇒ **Note:** Built-In Function Rules.

Sets a function for light usage mode.

For use with

LANSA for i YES

Visual LANSA for Windows NO

Visual LANSA for Linux NO

Arguments

No Argument Values.

Return Values

No Return Values.

Technical Notes

- This Built-In Function allows a function to dynamically change from heavy usage to light usage (and vice-versa). For example:

```
if (*jobmode = B)
    use SET_FOR_HEAVY_USAGE
else
    use SET_FOR_LIGHT_USAGE endif
```

- At every entry or (re)entry the heavy/light usage option is set by the calling parent process, or adopted from a *DIRECT caller, so at every invocation you should positively (re)set the usage option.
- This built-in executes very quickly and imposes little overhead.

9.218 SET_SESSION_VALUE

⇒ **Note:** Built-In Function Rules.

Allows various Visual LANSA session values to be dynamically altered by an executing application.

Note that a change to a session value only exists until the session (i.e., the X_RUN command) completes execution.

More information can be found on some session values by searching in the LANSA Global guide. In particular, refer to the [Standard X_RUN Parameters](#).

For use with

LANSA for i	YES	*Only USER_AUDIT session value may be used on IBM i in an RDML function.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of session value to be set or reset. Currently the following session values may be set or reset by this Built-In Function: USER: The current user - cannot be changed on IBM i. When you set USER, USEX (default user id when connecting to servers) also set. Refer to Technical Notes for Session Value USER=, GUSR= and USEX GUSR: The current group user - cannot be changed on IBM i.	1	50		

USER_AUDIT: The user to be used in user audit stamping. This must be explicitly set on the IBM i. Changing it in Visual LANSA will not update the IBM i in a Superserver application.

PRTR: The current printer

PPTH: The current fully qualified directory for printer files

JOBPTY: Job queue priority

DBMS_OPTIMIZE: Optimize heavy DBMS activity

CONNECT_PARTITION: The default partition to be used by the CONNECT_SERVER Built-In Function.

DPTH: Directory in which emulated IBM i data queues are to be created / accessed.

EXPS: The action to take when importing object security records.

HELP: Help system to use

HLPF: Help file to use

HMJB: Identifies that the current application is a Host Monitor job. 'Y' or 'N' . It is reserved for use by the LANSA development team only.

LANG: Change currently executing language. This is designed to be used only with intimate knowledge of the internal working of LANSA so use is restricted to applications created by the LANSA development team.

All tracing parameters e.g. ITRO. This can be useful for turning tracing on only whilst an issue is occurring so that the trace file size is kept to a minimum.

PSRA=: Primary Server Route Authority. Setting PSRA=Y indicates that authority checks should be routed to the server.

Use this command to set the value on the fly. This should be done before you define and make the connection. (Refer to PSRA Notes in [Using the X_RUN Command.](#))

PSRR: Primary Server Route Repository. Setting PSRR=Y (the default) indicates that if the repository data cannot be retrieved locally, a request should be sent to the server to retrieve the data.

Use this command to set the value on the fly. This should be done before you define and make the connection. (Refer to the X_RUN parameters in [Using the X_RUN Command.](#))

UDEF: User-defined values with spaces up to 256 bytes. If the value contains spaces or you wish to use embedded quotes, it must be enclosed in double quotes, reducing the usable length to 254 bytes.

Note:

All embedded quotes

must

be single quotes; if you embed double quotes it will cause submitted jobs to fail.

Windows printing extension parameters such as WPEN, WPPN, WPPS, WPPD, WPFD, WPDF, WPDS, WPFO and WPAS may be specified in this parameter. Any value specified in argument 2 should follow the same rules as defined for use of the windows printing extension parameter on the command line. (Refer to [Windows Printing Extensions.](#))

XCMD: Obscures values of specific parameters such as password. This parameter only becomes active for jobs submitted after it has been set.

CIPH: Symmetric cipher to use when LANSA calls OPenSSL.

2	A	Req	Value to which the session value should be set or reset. Refer to the following technical notes for more details.	1	256		
---	---	-----	---	---	-----	--	--

Return Values

No return values.

General Technical Notes

- The session value name (e.g., USER) that you specify is converted to uppercase before being tested for validity.
- The session value can be specified with or without the equal sign. For example USER= and USER are both valid and identical.
- The actual value that you specify may be converted to uppercase. Refer to the session value specific notes for details.
- Any value specified must not be blank or null. In all other cases it is accepted. You are responsible for ensuring the integrity and correctness of any value that you pass to this Built-In Function. Invalid or inappropriate specification of values may result in application failure and/or unexpected results on the current platform or any associated server platforms.
- Generally this Built-In Function is designed to be used very high in any invocation stack as an 'entry point' function that establishes various settings and that passes control into the actual application.
- Once a session setting is altered, it remains in effect until the session (ie: x_run command) completes execution. An altered session value effects all subsequently invoked facilities.
- When a session value is altered, functions, OAMs or communication sessions that are already active may ignore the change. As a very simple example of the concept of 'already active' consider:

```
define #chguser reffld(#user)
```

```
begin_loop
```

```
request #chguser
```

```
use set_session_value with_args('user=' #chguser)
```

```
display #user
```

```
end_loop
```

This function will not display the changed user. It will display #user (defaulted from *USER) at the time the function was originally invoked. The defaulting system variable *USER is static and is only initialized during function startup.

- The following specific session value technical notes may refer to a 'designated use'. When a designated use has been specified you must not use this Built-In Function in any other context.
- Invalid or inappropriate use of this Built-In Function may result in application failure and/or unexpected results on the current platform or any associated server platforms.

Technical Notes for Session Value USER=, GUSR= and USEX

- This session value is designated for use in a top level entry point function to allow you to design and implement your own 'sign on' function. After the user profile has been established the current user profile and group profile values may be altered.
- It is not designed to support multiple changes of user scattered throughout an executing application. Do not attempt to design or implement applications using such an approach.
- The value specified for USER= or GUSR= must conform to the rules defined in the x_run command. The correct value for no associated group user profile is *NONE.
- The value specified is always converted to uppercase.
- When you set USER, you also set USEX. The value you pass must be in a variable that maintains case. That value is first used to set USEX and then it is uppercased and USER is set.

USEX is the default User ID used when establishing connections to servers. For some servers the case is important and this allows the default to be changed in one place whilst also correctly controlling security access, if it is used. When the User ID is passed to X_RUN, it puts the exact case in the

USEX parameter, the same as this Built-In Function now does.

Technical Notes for Session Value USER_AUDIT

- This session value may be used to set the user that will be used in user audit stamping. It may be used where the current user does not give sufficient differentiations, such as in web jobs where all user jobs run under the default web user profiles.
- This session value is designated for use in a top level entry point function.
- It is not designed to support multiple changes of user audit scattered throughout an executing application. Do not attempt to design or implement applications using such an approach.
- The value can be retrieved via system variable *USER_AUDIT.
- No conversion to upper case is done on the value.
- The audit user setting remains in effect until the session completes execution.
 - In Visual LANSA (on Windows, IBM i RDMLX and Linux), a session is until the x-run command completes.
 - On IBM i RDML, the session value remains in effect until the job ends or the LANSA partition is changed.
 - In LANSA for the web jobs, the session value should be set for each browser interaction, since jobs may time-out and different jobs may be used to process a request.
 - Batch jobs start a new session. Consequently, the audit user should be set for the batch job session.
- If *LONG_USER_AUDIT is ON, then 256 characters of the session value will be used to set the value that will be used in user audit stamping. Otherwise 10 characters of the session value will be used to set this value.

Technical Notes for Session Value PRTR= and PPTH=

- The PRTR= session value is designated for use in reporting functions that wish to alter the name of the logical printing device to which a report should be directed.
- The PRTR= session value may also be used in reporting functions that wish to have all report output directed to a file. The special value *PATH should be specified and the PPTH= session value also used.
- The value specified for PRTR= must conform to the rules defined in the x_run command.

- The value specified for PPTH= must conform to the rules defined in the x_run command.
- The value specified is not converted to uppercase. This may be an important consideration for Linux systems.

Technical Notes for Session Value JOBPTY=

- This session value is designated for use in controlling the queuing priority of jobs submitted from this (ie: the currently executing) job. This value is defaulted to 5 when the x_run environment is invoked. This value should only be used with (and only has meaning in the context of) IBM i batch job queue and subsystem emulation.
- The value specified for JOBPTY= must be an integer value in the range 0 -> 9. Jobs submitted with the lowest JOBPTY value are executed from an emulated job queue first. Jobs with equal JOBPTY values are executed in arrival order.
- Please read the section in the *Deploying LANSAs Client and Server Applications* guide that deals with [IBM i Job Queue Emulation](#) before attempting to use this facility.

Technical Notes for Session Value CONNECT_PARTITION=

- This session value is designated for use in altering the partition identifier that the next CONNECT_SERVER Built-In Function invocation will attempt to connect to on the server system. The session value must be set prior to calling the builtin-in function CONNECT_SERVER and the value remains in effect until the X_RUN command ends or another CONNECT_PARTITION value is specified via this Built-In Function.
- This session value specified in the value argument (argument 2) should be the 3 character identifier of a valid partition on the server system.
- The value specified is always converted to uppercase.

Technical Notes for Session Value DBMS_OPTIMIZE=

Portability When using this Built-In Function in RDMLX code on the
Considerations IBM i, a COMMIT is issued so Commitment Control must be started.

- The DBMS_OPTIMIZE session value is designed to allow you to specify how a heavy or complex DBMS transaction can be optimized.

Judicious use of this option can significantly improve the performance of

heavy or complex DBMS activity.

The value parameter, which must be in uppercase, may be set to:

BEGIN_SYNC_nnnn Specifies the beginning of a complex DBMS activity and switches on DBMS optimize mode. "Hard" commits are to be performed at every "nnnn" invocations of the "soft" synchronization point.

SYNC_POINT Specifies a "soft" synchronization point for a complex DBMS activity. A soft synchronization point is the equivalent of a "optimized" commitment control boundary.

END_SYNC Specifies the end of a complex DBMS activity and switches off DBMS optimize mode.

The use of DBMS_OPTIMIZE is best illustrated by example.

Consider this simple RDML function:

```
begin_loop from(1) to(2000)
    insert fields(.....) to_file(testfile)
end_loop
```

In normal circumstances this function would be using auto-commit and would be performing a DBMS commit operation after every single insert.

Most DBMS commit operations are usually quite slow. The commit can have a very significant performance impact on this type of application (ie: one doing the same thing 1000's rather than just 5 or 10 times).

If the function was changed to:

```
use set_session_value ( DBMS_OPTIMIZE BEGIN_SYNC_100 )
```

```
begin_loop from(1) to(2000)
  insert fields(.....) to_file(testfile)
  use set_session_value ( DBMS_OPTIMIZE SYNC_POINT )
end_loop
```

```
use set_session_value ( DBMS_OPTIMIZE END_SYNC )
```

then it may run much faster than before. The DBMS_OPTIMIZE operations are enabling the underlying OAM (Object Access Module) to understand the transaction better, and thus to optimize it.

The primary reason for the performance improvement is the fact that the "soft" SYNC_POINT will only issue "hard" (ie: real) DBMS commit operations every 100 times that it is executed. This is why the BEGIN_SYNC_100 value is used. Thus the whole DBMS transaction begins to work faster.

Another example might be this "batch" style function:

```
select fields(.....) from_file(customer) where(.....)
  select fields(.....) from_file(orders) with_key(customer)
    call *direct calculate
  select fields(.....) from_file(items) with_key(orderno)
    update fields(.....) in_file(items)
endselect
  update fields(.....) in_file(orders)
endselect
```

endselect

The performance of this batch style program, could be enhanced by changing it to this:

```
use set_session_value ( DBMS_OPTIMIZE BEGIN_SYNC_20 )
```

```
select fields(.....) from_file(customer) where(.....)
```

```
select fields(.....) from_file(orders) with_key(customer)
```

```
call *direct calculate
```

```
select fields(.....) from_file(items) with_key(orderno)
```

```
update fields(.....) in_file(items)
```

endselect

```
update fields(.....) in_file(orders)
```

endselect

```
use set_session_value ( DBMS_OPTIMIZE SYNC_POINT ) endselect
```

```
use set_session_value ( DBMS_OPTIMIZE END_SYNC )
```

Some things that you must understand and rules that you must follow when using DBMS_OPTIMIZE are:

BEGIN_SYNC_nnnn

- Must be specified in uppercase characters.
- Will abort if a BEGIN_SYNC_nnnn is already active within the current job. BEGIN_SYNC_nnnn operations cannot be nested.
- Sets a synchronization "trigger" to the "nnnn" value specified. "nnnn" must be a valid integer in the range 1 to 1000.

Setting "nnnn" to 1 may negate any possible performance improvement because the SYNC_POINT will issue a commit operation every time it is executed.

- Set a synchronization point "counter" to zero.
- Switches on DBMS optimize mode.
- Switches off all "auto-commitment" options. All DBMS tables will now be under full manual commitment control, regardless of how they are defined. Manual commits are issued when:
 - A SYNC_POINT reaches a trigger level.
 - An END_SYNC is executed.
 - A COMMIT operation is issued by an RDML function.
- Must have an associated END_SYNC operation that is always executed to indicate the normal completion of a DBMS transaction.
- May have an associated SYNC_POINT operation that specifies the point of a commitment control boundary. If no SYNC_POINT is specified then the END_SYNC operation acts as the single commitment control boundary.
- Generally the higher the "nnnn" value the better the performance (within DBMS and journalling limits), however values above 500 would be unusual. Remember that the higher the value the more system resources being held/locked and the more data that will be lost if a rollback event occurs.
- Should be used with SYNC_POINT and END_SYNC in a simple and well structured way to establish commitment boundaries and to optimize performance. Avoid scattering BEGIN_SYNC, SYNC_POINT and END_SYNC operations across multiple functions and down through complex invocations stacks. Try to keep all operations in a single function and keep the boundaries simple and well delimited.
- Should not be used in complex and convoluted invocation stacks involving many different RDML functions and function calls. The primary reason for this is the very real possibility that a called function may well issue a COMMIT (or perform some action that causes a commit to be performed) thus negating any performance benefit that this option may bring.
- Has no real meaning when used in LANSA SuperServer mode.
- It is recommended that jobs concurrently accessing more than 15 tables should not use this facility.

SYNC_POINT

- Must be specified in uppercase characters.
- Will abort if DBMS optimize mode is not currently turned on.
- Increments the synchronization point "counter"
- If the synchronization "counter" is greater than or equal to the synchronization "trigger" value, then a DBMS commit operation is performed and the counter is reset to zero.
- Will still issue a commit when used in LANSA SuperServer mode.

END_SYNC

- Must be specified in uppercase characters.
- Will abort if DBMS optimize mode is not currently turned on.
- Unconditionally issues a DBMS commit.
- Resets the synchronization counter to zero.
- Resets the synchronization trigger to a default value.
- Turns DBMS optimize mode off.

Will still issue a commit when used in LANSA SuperServer mode.

9.219 SHOW_HELP

⇒ **Note:** Built-In Function Rules.

Causes the help text associated with a field, a component, a process or a function to be modally displayed. .

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object Type. Specify this argument as: PD – Process PF – Function DF – Field or Component Any other value is invalid and will cause a run time error to be generated.	2	2		
2	A	Req	Object Name. Blank or null object names will cause a run time error to be generated. Other invalid or non-existent object names may cause help text to be displayed indicating that no help text exists.	1	10		
3	A	Opt	Object Name Extension. Specify this argument only for type PF (Function) or DF (Component only, NOT Field). For a Function it is the name of the process to which the function belongs and it must be no	1	65		

		longer than 10 characters. For a Component it is the name of the specific Component within the owning Component. For all other object types this argument is ignored, regardless of what value it contains.				
--	--	---	--	--	--	--

Return Values

There are no return values

Comments

- The help text associated with a component should only be displayed in a component context (i.e. from an RDMLX program).
- Field, Process and Function help text can be displayed in any context (i.e. from RDMLX programs or RDML functions).
- In this version of this Built-In Function the display of help text is modal. The application halts until the user completes the display of the help text and closes the help text window.
- In future versions of this Built-In Function the display of help text may not be modal. You should avoid designing applications that architecturally rely on the modal nature of the current version.
- This Built-In Function should never be executed in environments in which no user interface capability exists (e.g. in a trigger or in a remote procedure).

Examples

Display the help text associated with field EMPNO:

```
USE BUILTIN(SHOW_HELP) WITH_ARGS(DF EMPNO);
```

Display the help text associated with process MYPROC:

```
USE BUILTIN(SHOW_HELP) WITH_ARGS(PD MYPROC);
```

Display the help text associated with the TREEVIEW component in the component VL_DEM20:

```
USE BUILTIN(SHOW_HELP) WITH_ARGS(DF VL_DEM20  
TREEVIEW);
```

9.220 SND_TO_DATA_QUEUE

⇒ **Note:** Built-In Function Rules.

Places one or more entries from a working list onto an IBM i or Windows emulated data queue. Refer to IBM supplied manuals for more details of data queues.

Note: Only use this Built-In Function in applications that are to fully execute under the control of the IBM i or a Windows operating system.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	A literal or variable that specifies or contains the name of the data queue. This name must conform to IBM i object naming conventions. This is not checked by the Built-In Function.	1	10		
2	N	Req	A literal or variable that specifies or contains the byte length of one complete entry of the working list specified in argument 3.	1	5	0	0
3	L	Req	The name of the working list whose entries are to be sent to the specified data queue.	1	10		

Return Values

No Return Values.

Technical Notes - General

- When the working list contains multiple entries, **all** entries are sent to the data queue in **one** operation. For example, if a working list had an entry length of 20 bytes, and it contained 7 entries, then the actual length of the data record sent to the data queue would be $20 * 7 = 140$ bytes. This processing must be considered when specifying the MAXLEN parameter of the IBM supplied CRTDTAQ (create data queue) command under IBM i.
- If a list is passed to the Built-In Function that contains zero (0) entries no error occurs and no data is written to the data queue.
- For maximum speed, no significant checking is performed by this Built-In Function. Errors with lengths or data types may cause failures that require detailed analysis of error information.
- The backup, recovery and maintenance of any data queues accessed via this Built-In Function is entirely the responsibility of the user.
- Data queues are persistent objects but their data content can be lost or corrupted during a system failure.
- Data queues are limited in how much information they can store. Typically no more than 16Mb of data should be queued at any point in time.

Technical Notes - IBM i Operating System

- The data queue specified must be created by using the IBM supplied CRTDTAQ (create data queue) command.
- Data queues may be cleared by using the IBM supplied API QCLRDTAQ.
- Data queues may be deleted by using the IBM supplied DLTDTAQ (Delete Data Queue) command.
- The data queue must be able to be located in the job's current library list. Support for fully qualified library/queue names is not provided in the interests of good design practice.

Technical Notes - Windows Operating Systems

- Data queues are automatically created when they are referenced.
- Data queues may be cleared by deleting the file used to store the data queue

data. They will be automatically (re)created the next time they are referenced.

- Data queues may be deleted by deleting the file used to store the data queue data and the file used to control data queue locking.
- Data queues are stored in a normal Windows file.
- The data queue storage file name is an 8 character conversion of the data queue name. The conversion process uses the same algorithm as is used to convert 10 character LANSAs process names to an 8.3 formatted DLL name.
- The data queue storage files are suffixed by .EDQ (for Emulated Data Queue) and .LDQ (for Lock Data Queue). The .LDQ file will only exist if a function has at some time attempted a receive operation from the queue.
- The .EDQ file stores the actual queue data. Space in this file is reused as queue entries are removed, thus the size of this file represents a high water mark.
- The .LDQ file is used to logically lock a data queue during receive operations. This file can be deleted at any time as it is automatically (re)created on demand.
- By default data queue storage files are created in the <sysdir>\x_ppp directory (where ppp is the partition) of the current LANSAs environment.
- The location of the data queue storage files can be controlled by using the DPTH= parameter of the X_RUN command. For example, DPTH=c:\temp would cause all data queues to be created and accessed in the c:\temp directory.
- The DPTH= parameter value can be dynamically altered in an application by using the SET_SESSION_VALUE Built-In Function.
- You should not use the DPTH= parameter for any purpose other than the simple one time redirection of all data queue accesses to an alternative directory. Complicated designs that use many instances a single data queue name in many different directories should be avoided (for the same reason that support for IBM i fully qualified library/queue names is not provided).

Examples

Receive a customer number and a part number from a screen panel and then place them onto a data queue called PICKLIST:

```
DEF_LIST NAME(#PICK) FIELDS(#CUSTNO #PARTNO) TYPE(*WORK  
ENTRYS(1)
```

(where #CUSTNO is defined in the dictionary as a signed 5,0 number and
 REQUEST FIELDS(#CUSTNO #PARTNO)
 INZ_LIST NAMED(#PICK) NUM_ENTRYS(1)
 USE BUILTIN(SND_TO_DATA_QUEUE) WITH_ARGS('PICKLIST' 9 #)

Receive 5 customer numbers and part numbers from a screen panel and then
 place them onto a data queue called PICKLIST as 5 **separate** data queue
 entries:

```

DEF_LIST NAME(#PICK) FIELDS(#CUSTNO #PARTNO) TYPE(*WORK
BEGIN_LOOP FROM(1) TO(5)
  REQUEST FIELDS(#CUSTNO #PARTNO)
  INZ_LIST NAMED(#PICK) NUM_ENTRYS(1)
  USE BUILTIN(SND_TO_DATA_QUEUE) WITH_ARGS ('PICKLIST' 9 #)
END_LOOP

```

Receive 5 customer numbers and part numbers from a screen panel and then
 place them onto a data queue called PICKLIST as a **single** data queue entry:

```

DEF_LIST NAME(#PICK) FIELDS(#CUSTNO #PARTNO) TYPE(*WORK
CLR_LIST NAMED(#PICK)
BEGIN_LOOP FROM(1) TO(5)
  REQUEST FIELDS(#CUSTNO #PARTNO)
  ADD_ENTRY TO_LIST(#PICK)
END_LOOP
USE BUILTIN(SND_TO_DATA_QUEUE) WITH_ARGS('PICKLIST' 9 #)

```

9.221 SPACE_OPERATION

⇒ **Note:** Built-In Function Rules.

Requests a miscellaneous space object operation.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name	1	256		
2	A	R	Requested Operation	1	256		
3- 20	X	O	Fields that specify additional operation dependent arguments.	1	Unlimited	0	Unlimited

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "ER" = Request failed. Messages issued will indicate more about the cause of the failure. Other return code values depend	2	2		

			upon the operation requested. See the description of each operation for more details.				
2-20	X	O	Fields that specify additional operation dependent return values.	1	Unlimited	0	Unlimited

The CheckExistence Operation

The CheckExistence operation can be used to check for the existence of a space object within the current operating system process (or job).

In the following example, it checks for the existence of a space named TestSpace:

```
Use Builtin(SPACE_OPERATION) With_Args('TestSpace' CHECKEXISTENCE
```

The return code is only ever returned as "OK" (space object exists) or "NR" (space object does not exist) by valid CheckExistence requests.

The SetCursor Operation

The SetCursor operation can be used to set the cursor for ALL subsequent invocations of the SELECT_IN_SPACE or SELECTNEXT_IN_SPACE BIFs. It is not specific to the space object. If the cursor is set to 2, all subsequent select BIFs for any space object will use cursor 2 until it is changed to something different. The default cursor is 1, and up to 4 cursors can be specified for each space object. The only checking performed is to ensure that the cursor number is between 1 and 4.

For example ...

```
Use Builtin(SPACE_OPERATION) With_Args('TestSpace' SETCURSOR 2) To_
```

The return code is only ever returned as "OK" (space object exists) or "ER" (cursor number invalid) by valid SetCursor requests.

This operation permits SELECT_IN_SPACE requests for the same space object

to be nested. To be used effectively, the SetCursor operation should be called just before using the SELECT_IN_SPACE or SELECTNEXT_IN_SPACE BIFs to ensure that the correct cursor is being used. This is typical of the use of the SetCursor operation to ensure that no unwanted side-effects occur by using the wrong cursor because, for example, the LEAVE command was issued from the inner loop:

```
Use Builtin(SPACE_OPERATION) With_Args(#AR_SPACE SetCursor 1) To_
Use Builtin(SELECT_IN_SPACE) With_Args(#AR_SPACE) To_Get(#SPACE
Dowhile Cond(*SPACEOK)
```

```
Use Builtin(SPACE_OPERATION) With_Args(#AR_SPACE SetCursor 2) T
```

* Note the use of a different set of fields to hold the data - #XG_AR2

```
Use Builtin(SELECT_IN_SPACE) With_Args(#AR_SPACE) To_Get(#SPA(
Dowhile Cond(*SPACEOK)
```

* Do something with the data here

```
Use Builtin(SPACE_OPERATION) With_Args(#AR_SPACE SetCursor 2)
Use Builtin(SELECTNEXT_IN_SPACE) With_Args(#AR_SPACE) To_G
Endwhile
```

```
Use Builtin(SPACE_OPERATION) With_Args(#AR_SPACE SetCursor 1) T
Use Builtin(SELECTNEXT_IN_SPACE) With_Args(#AR_SPACE) To_Get(
Endwhile
```

* Set the cursor back to the default for any further SELECT... BIF calls.

```
Use Builtin(SPACE_OPERATION) With_Args(#AR_SPACE SetCursor 1) To_
```

9.222 SQUARE_ROOT

⇒ **Note:** Built-In Function Rules.

Calculates a square root value.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Value to have square root calculated. Note: In Visual LANSA the maximum length is 15,5	1	30	0	9

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Square root value. Note: In Visual LANSA the maximum length is 15,5	1	30	0	9

Note: This function will truncate the root value into the return value. That is, if the return field is defined as digits(4) decimals(2) and the value of the square root is 1.41421, the value returned will be 01.41.

Examples

******* A simple square root calculator**

```
DEFINE FIELD(#NUMBER1) TYPE(*DEC) LENGTH(15) DECIMALS(5)
```

```
DEFINE FIELD(#RESULT) TYPE(*DEC) LENGTH(15) DECIMALS(5) E
```

******* Request number to calculate root from**

```
REQUEST FIELDS(#NUMBER1)
```

******* Calculate square root**

```
USE BUILTIN(SQUARE_ROOT) WITH_ARGS(#NUMBER1) TO_GET
```

******* Display result**

```
DISPLAY FIELDS(#NUMBER1 #RESULT)
```

9.223 START_FILE_EDIT

⇒ **Note:** Built-In Function Rules.

Starts an "edit session" on the definition of a nominated LANSAs file definition.

The edit session can be used to define a new file or alter an existing one.

The file **definition** is locked for exclusive use throughout the edit session.

Only one file definition can be edited at one time (ie: it is not possible to concurrently edit 2 file definitions from within the same job).

Details of the new or amended file definition will be lost unless the END_FILE_EDIT Built-In Function is used to "commit" them.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of file to be edited	1	10		
2	A	Req	Library in which file resides *LIBL or *FIRST is acceptable when editing an existing file definition. *DEFAULT is acceptable when editing an existing file definition or creating a new one. Note: When coding *FIRST, *DEFAULT,	1	10		

			<p>*LIBL,etc., they must be placed in extra quotes to distinguish them as literals and not system variables. Thus, three single quotes must be coded round the argument. e.g. <code>""*FIRST"</code>.</p> <p>It is NOT valid to code double quotes and single quotes (<code>""*FIRST"</code>)</p> <p>Refer to the Example.</p>				
3 **	A	Req	Source of edited details. Must not be blank, LAN or OTH.	3	3		
4	A	Opt	File description Must not be blank. Required for a new file.	1	40		
5	A	Opt	Initial public access. Required for a new file. ALL, NORMAL or NONE allowed.	1	6		
6	A	Opt	<p>File component edit options</p> <p>Byte 1: Y or N indicates that fields are to be edited. Default value is Y.</p> <p>Byte 2: Y or N indicates that logical views/files are to be edited. Default value is Y.</p> <p>Byte 3: Y or N indicates that access route details are to be edited. Default value is Y.</p> <p>When a byte is passed as N, the associated component of the file definition remains unchanged during the edit session and is not flagged for pending deletion as described below. Any attempt to edit that component of the file definition will cause a fatal error.</p>	3	3		

** The "source" of the edited details is vital. When an edit session is commenced all details of the file definition that have the same "source" as that passed to the START_FILE_EDIT Built-In Function are flagged for pending deletion. If the details are not "re-specified" by one of the FILE_FIELD,

LOGICAL_VIEW, etc Built-In Functions, they are deleted from the file definition by the END_FILE_EDIT Built-In Function. The exception is those that have been specifically excluded from editing using one or more of the byte positions in the 6th argument previously described. This allows for file details specified by other sources (such as direct input via the LANSAs Review or change a file definition facility in the [The File Control Menu](#)) to remain intact during a file edit session.

Examples of this "source" code would be:

LDM LANSAs data modeling interface

IEW Information engineering workbench interface

ACC Accelerate data modeling interface

Once set, the source code used should never be changed within a particular type of interface.

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = edit session commenced ER = fatal error detected In case of "ER" return code error message(s) are issued automatically and the edit session ended without commitment.	2	2		

Example

The following example defines all details of a simple name and address file called NAMES to the LANSAs system by using Built-In Functions rather than the conventional menu driven interfaces.

```
*** Define the fields into the data dictionary (no prompting)
```

```
*
```

```
USE BUILTIN(PUT_FIELD) WITH_ARGS('N' 'CUSTNO' 'S' 007 0 ' '
'Customer number') TO_GET(#RETCODE)
```

```

USE  BUILTIN(PUT_FIELD) WITH_ARGS('N' 'CUSNAME' 'A' 010 0 ''
    'Customer name') TO_GET(#RETCODE)
USE  BUILTIN(PUT_FIELD) WITH_ARGS('N' 'ADDRESS1' 'A' 020 0 ''
    'Address line 1') TO_GET(#RETCODE)
USE  BUILTIN(PUT_FIELD) WITH_ARGS('N' 'ADDRESS2' 'A' 020 0 ''
    'Address line 2') TO_GET(#RETCODE)
USE  BUILTIN(PUT_FIELD) WITH_ARGS('N' 'ZIPCODE' 'S' 006 0 ''
    'Zip code') TO_GET(#RETCODE)
*
*** Start an edit session on the new file NAMES in library QGPL
*
USE  BUILTIN(START_FILE_EDIT) WITH_ARGS('NAMES' 'QGPL' 'DEI
    'Customer details' 'NORMAL') TO_GET(#RETCODE)
*
*** Use of triple quotes round *FIRST and *DEFAULT libraries
*
USE  BUILTIN(START_FILE_EDIT) WITH_ARGS('CODES' ""*FIRST"" 'T
*
USE  BUILTIN(START_FILE_EDIT) WITH_ARGS('SALES' ""*DEFAULT
*
*** Define the fields in the file
USE  BUILTIN(FILE_FIELD) WITH_ARGS('CUSTNO') TO_GET(#RETC
USE  BUILTIN(FILE_FIELD) WITH_ARGS('CUSNAME') TO_GET(#RE
USE  BUILTIN(FILE_FIELD) WITH_ARGS('ADDRESS1') TO_GET(#RE
USE  BUILTIN(FILE_FIELD) WITH_ARGS('ADDRESS2') TO_GET(#RE
USE  BUILTIN(FILE_FIELD) WITH_ARGS('ZIPCODE') TO_GET(#RETC
*** Define the primary or relational file key
*
USE  BUILTIN(PHYSICAL_KEY) WITH_ARGS('CUSTNO') TO_GET(#R
*
*** Define additional logical view in CUSNAME / ZIPCODE order
USE  BUILTIN(LOGICAL_VIEW) WITH_ARGS('NAMESV1' 'Customers
*
*** Define keys of logical view NAMESV1
USE  BUILTIN(LOGICAL_KEY) WITH_ARGS('NAMESV1' 'CUSNAME'
USE  BUILTIN(LOGICAL_KEY) WITH_ARGS('NAMESV1' 'ZIPCODE')

*** Define "one to one" access route to ZIPTABLE by using key ZIPCODE
USE  BUILTIN(ACCESS_RTE) WITH_ARGS('DEM1' 'Zip details' 'ZIPTAI

```

```
USE BUILTIN(ACCESS_RTE_KEY) WITH_ARGS('DEM1' 'ZIPCODE') T
*
*** Define "one to many" access route to ORDHDRV2 using key CUSTNO
USE BUILTIN(ACCESS_RTE) WITH_ARGS('DEM2' 'Order details' 'ORD
USE BUILTIN(ACCESS_RTE_KEY) WITH_ARGS('DEM2' 'CUSTNO') T
*
*** End the edit session and commit details
USE BUILTIN(END_FILE_EDIT) WITH_ARGS('Y') TO_GET(#RETCOL
```

9.224 START_FUNCTION_EDIT

⇒ **Note:** Built-In Function Rules.

Starts an "edit session" on the definition of a nominated LANSAs function definition.

The edit session can be used to define a new function or alter an existing one.

A function edit session must be started and ended within a process edit session on the parent (i.e.: owning) process. Multiple edit sessions on functions within the same process may be conducted serially (but not concurrently) within the same process edit session.

For example:

```
START_PROCESS_EDIT
```

```
    START_FUNCTION_EDIT
```

```
    << work with function A >>
```

```
    END_FUNCTION_EDIT
```

```
END_PROCESS_EDIT
```

or:

```
START_PROCESS_EDIT
```

```
    START_FUNCTION_EDIT
```

```
    << work with function A >>
```

```
    END_FUNCTION_EDIT
```

```
    START_FUNCTION_EDIT
```

<< work with function B >>

END_FUNCTION_EDIT

END_PROCESS_EDIT

The function definition is locked for exclusive use throughout the function edit session.

Only one function definition can be edited at one time (ie: it is not possible to concurrently edit two or more function definitions within the same job).

A function edit session should be terminated by using the END_FUNCTION_EDIT Built-In Function to ensure all locks/etc are released/shutdown in an orderly manner.

Any function edit session that receives a fatal error will have an END_FUNCTION_EDIT and an END_PROCESS_EDIT operation automatically issued.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req	Description	Min	Max	Min	Max
----	------	-----	-------------	-----	-----	-----	-----

		Opt		Len	Len	Dec	Dec
1	A	Req	Name of function to be edited	1	7		
2	A	Opt	Function Description. Required for a new function only. Must not be blank. Default value is FUNCTION.	1	40		
3	A	Opt	Initial public access. Required for a new function only. ALL, NORMAL or NONE allowed. Default value is NORMAL.	3	6		
4	A	Opt	Include RDML audit stamps (Y,N). Controls the content and length of the RDML working lists used in the following Built-In Functions in this function edit session: GET_FUNCTION_RDML EXECUTE_TEMPLATE and PUT_FUNCTION_RDML Y=RDML audit stamping is included in the RDML working lists used in this function edit session. N=RDML audit stamping is not included in the RDML working lists used in this function edit session. Default value is N.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = edit session commenced ER = fatal error detected	2	2		

9.225 START_PROCESS_EDIT

Starts an "edit session" on the definition of a nominated LANSAs process definition.

The edit session can be used to define a new process or alter an existing one. The process definition is locked for exclusive use throughout the edit session. Only one process definition can be edited at one time (ie: it is not possible to concurrently edit two or more process definitions within the same job).

A process edit session should be terminated using the END_PROCESS_EDIT Built-In Function to ensure all locks/etc are released/shutdown in an orderly manner.

Any process edit session that receives a fatal error will have an END_PROCESS_EDIT command automatically issued.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of process to be edited	1	10		
2	A	Opt	Process description.	1	40		

			Used for a new process only. Default value is PROCESS.				
3	A	Opt	Process menu type / style. Used for a new process only. Must be SAA/CUA or ACT/BAR. Default value is SAA/CUA.	7	7		
4	A	Opt	Initial public access. Used for a new process only. ALL, NORMAL or NONE allowed. Default value is NORMAL.	3	6		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = edit session commenced ER = fatal error detected	2	2		

9.226 START_RTV_SPLF_LIST

⇒ **Note:** Built-In Function Rules.

Used in conjunction with GET_SPLF_LIST_ENTRY and END_RTV_SPLF_LIST. It must be used first to provide the selection criteria for the retrieval of spool files. The selection criteria which can be specified are User Name, Output queue name and library, Form Type, User Data and Status. Once this START_RTV_SPLF_LIST has been used to establish the selection, the GET_SPLF_LIST_ENTRY can be used to retrieve the details of the spool files.

The END_RTV_SPLF_LIST must be used after the list of spool files have been retrieved. This will close list and release the storage allocated to that list.

Refer to [9.130 GET_SPLF_LIST_ENTRY](#) for an example.

For use with

LANSA for i	YES	Not available for RDMLX.
Visual LANSA for Windows	NO	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	User Name This argument should contain the User Name of the spool files you want to retrieve. A value of *ALL will select all users. Default value is *ALL.	1	10		
2	A	Opt	Output Queue Name This argument is used in conjunction with Output Queue Library to determine from	1	10		

			which output queue to retrieve spool files. A value of *ALL will select all output queues. Default value is *ALL.				
3	A	Opt	Output Queue Library This argument is used in conjunction with Output Queue Name to determine from which output queue to retrieve spool files. Default value is *LIBL.	1	10		
4	A	Opt	Form Type This argument should contain the Form Type of spool files to retrieve. A value of *ALL will select all form types. Default value is *ALL.	1	10		
5	A	Opt	User Data This argument should contain the User Data of spool files to retrieve. A value of *ALL will select spool files regardless of their user data. Default value is *ALL.	1	10		
6	A	Opt	Status This argument should contain the Status of the spool files to select. Default value is *ALL. Possible values are *ALL, *CLOSED, *DEFERRED, *SENDING, *FINISHED, *HELD, *MESSAGE, *OPEN, *PENDING, *PRINTING, *READY, *SAVED, *WRITING.	1	10		

Return Values

No	Type	Req/	Description	Min	Max	Min	Max
----	------	------	-------------	-----	-----	-----	-----

		Opt		Len	Len	Dec	Dec
1	A	Opt	Return code. OK = The list was opened successfully. ER = the argument selection details are invalid. NR = no spool files were found which match the selection arguments.	2	2		

9.227 STM_FILE_CLOSE

This Built-In function closes the stream file which was previously opened by a STM_FILE_OPEN.

All stream files opened should be closed before terminating a function. Once the stream file is closed, no further action against that file is possible until it is re-opened.

Related Built-In Functions: STM_FILE_OPEN, STM_FILE_READ, STM_FILE_WRITE, STM_FILE_WRITE_CTL

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	File number (file handle) of the file that is to be closed. This number was allocated to a stream file and returned on the STM_FILE_OPEN.	1	3	0	0

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK = File has been closed	2	2		

		ER = Error occurred.				
--	--	----------------------	--	--	--	--

Example

Refer to [9.228 STM_FILE_OPEN](#).

9.228 STM_FILE_OPEN

⇒ **Note:** Built-In Function Rules.

This Built-In function opens or creates a stream file. Options control how the file is opened or created. Associated Built-In Functions may be used to read or write data to the stream file.

When executing in LANSAs for i, these Built-In functions may be used to read and write to IBM i IFS files.

Related Built-In Functions: [9.229 STM_FILE_READ](#), [9.230 STM_FILE_WRITE](#), [9.227 STM_FILE_CLOSE](#) and [9.231 STM_FILE_WRITE_CTL](#).

For use with

LANSAs for i	YES
Visual LANSAs for Windows	YES
Visual LANSAs for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Full path and file name of the file to be opened. eg /mydir/myfile.txt. See File Open Options following. If the file is opened for read, the file must exist. If the file is opened for write or append, the file will be created if it does not exist.	1	Unlimited		
2	A	Opt	Options that are to be applied to handling of this stream file.	1	Unlimited		

			<p>Multiple options may be specified. Options must be separated by at least one blank character. Refer to Technical Notes for details. Possible options: Read Write Append Text Binary LineTerminator=aaaaa LineTerminator=LF NoTrim CodePage=nnnnn</p>				
3	A	Opt	<p>Used on IBM i only: Authority level to set the public authorities when a file is created. N=NONE no authorities set A=ALL set authority to RWX R=READUSE set authority to RX Default value is R</p>	1	1		
4	A	Opt	<p>Buffering. This option is only valid if the file is opened for a Write or Append operation. If this option is set to N, data will be written to permanent storage before the BIF returns. Y = Buffering. N = No buffering. Default value is Y</p>	1	1		
5	A	Opt	<p>Used on IBM i only: When a file is created, set the file authorities to the Group Profile settings obtained from the User Profile. N= Not apply Y- Apply</p>	1	1		

		Default value is N.				
--	--	---------------------	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	File number (file handle) A unique file number is allocated to this file when opened and returned in this value. This file number must be used when further actions are made against this file. For example when data is to be read from this file using STM_FILE_READ this file number must be provided. Up to 256 files may be opened at any time. Any of these files may be processed by providing the appropriate file number.	1	3	0	0
2	A	Opt	Return Code OK = File has been opened and file number allocated. ER = Error occurred on the file open. Refer to messages for details.	2	2		

Technical Notes

File Open Options

Note: IBM i path separator is / (forward slash). \ (back slash) is not acceptable in IBM i. It will create an unreachable file.

Option group	Default value if not specified	Option values which may be specified	Notes
File mode	Read	Read	The file will be

		opened for read. The file must exist.
	Write	The file will be opened for write. If the file does not exist it will be created. If the file does exist the data contents will be overwritten.
	Append	The file will be opened for write at the end of the file. Write actions will append the data to the existing file. If the file does not exist it will be created.
Data mode	Text	Text
		Data is read/written as a text stream. Character conversion is done. The data will be converted between the code page of the job and the code page of the file.
	Binary	Data is

		read/written as a binary stream. No character conversion is done. The data is not altered on input or output. If a file is created in Binary mode, it will be created with the code page of the job.
<p>Line format LineTerminator=LF</p> <p>This option enables you to specify the line terminating character/s.</p> <p>On read files, this character is searched for and data up to the terminator returned by each read action. The terminator character/s is/are not returned.</p> <p>On files being written the line terminator is added to the end of the data</p>	LineTerminator=ALL	Valid on READ files only. Any of the line terminators (CR, CRLF, LF, NL or LFCR) will be used to indicate the end of a line.
	Default=LF	
	LineTerminator=CR	Terminator character is Carriage Return
	LineTerminator=CRLF See Windows Text Mode Handling	Terminator characters are Carriage Return, Line Feed
	LineTerminator=LF See Windows Text Mode Handling	Terminator character is Line Feed
LineTerminator=NL	Terminator character is	

<p>on each write action.</p>			New Line
		<p>LineTerminator=LFCR See Windows Text Mode Handling</p>	<p>Terminator characters are Line Feed, Carriage Return</p>
		<p>LineTerminator=NONE All line terminators (LT) are stripped off the ends of lines after reading and before writing. The end of line is what the BIF has recognised as an end of line, not the file's end of line. If an LT is chosen that does not match the file, then no LTs will be stripped because they are not at the end of the line. Similarly on output. Each line that is written out by the BIF will have LT stripped from the end only.</p>	<p>No line terminator characters are used.</p>
<p>Trim trailing blanks This option may be used for files being written and controls the trimming of trailing blanks from Data Blocks provided to the write</p>	<p>The default action is to trim trailing blanks from Data Blocks before writing to the data stream.</p>	<p>Notrim</p>	<p>Trailing blanks will not be trimmed from Data Blocks provided on the write action.</p>

action.			
Code page This option is used when files are created . Files created on IBM i IFS will be created as the specified code page. This option is ignored when executing on a non IBM i platform.	The code page of the current job is used if a file is created.	CodePage=nnnnn where nnnnn is the code page required. Note: CodePage=00819 is usual for English IBM i IFS files.	This option will be ignored by a file opened for writing in Append mode to an existing file. The code page information is used only if the file needs to be created.

Windows Text Mode Handling

Text mode, when reading a file with 0x0d0a line termination Windows only returns 0x0a. To match this you need to specify LineTerminator=LF or ALL

In **Text mode**, when writing a file, 0x0d is output by Windows if 0x0a is in the buffer. Thus when LF is specified, the file gets 0x0d0a. When CR is specified file gets 0x0d. When CRLF is specified file gets 0x0d0d0a.

In **Binary mode** the 0x0d0a is read intact. Thus CRLF works.

When the stream file Built-In Functions are used to copy from one file to another and the line termination in the file is 0x0d0a on Windows, these combinations preserve the CRLF:

- **Read Text mode**, LineTerminator=LF or ALL, Write Text Mode, LineTerminator=LF
- **Read Binary mode**, LineTerminator=CRLF or ALL, Write Binary Mode, LineTerminator=CRLF

Also, unless there is special processing for flagging that reading a line has returned OV, extra lines will be output when writing. Either make your read

buffer greater than the maximum length or handle the OV lines when writing so that the line is re-constituted.

Binary mode is more portable than text mode because it is entirely predictable. It tells the OS to get out of the way and let LANSAs handle it. On Windows there is no character conversion in text mode so there is very little reason to use it—it only causes confusion. Use Binary on Windows. You know that the Built-In Function will receive your data exactly as you see it in the file and will output it exactly as you specify it.

End of Line markers vary from platform to platform. Windows uses 0x0d0a and Linux uses 0x0a.

Its probable that using Text mode and LineTerminator=LF for both Windows and Linux would also work.

Code page conversion

When executing on an IBM i platform, conversion between the code page of the executing job and the code page of the stream file may occur. The code page of the stream file is established when the file is created.

When reading data from a file in text mode, the data is converted from the file's code page to the job's code page. When reading data in binary mode, no conversion is done.

When a file is created for writing, it will be created with the code page specified on the STM_FILE_OPEN. If no code page was provided, it will default to the code page of the executing job. Text data written to a file will be converted from the code page of the job to the code page of the file. Data written in Binary mode will not be converted.

If a file is opened for writing in Append mode to an existing file, the code page of the existing file remains unchanged.

On Windows and Linux, no code page conversion occurs. When reading data from a file in text mode, the data is assumed to be in the current code page. Data in UTF-8 and UTF-16 is not supported.

Example

A stream file on the IBM i IFS is opened. The stream file is in directory /tmp and named updphone.txt . The file is to be read as text and any of the standard line terminators (CRLF, LFCR, CR, LF, NL) indicates the end of each line. Each line of the stream file is read and the information used to update database file PSLMST. When the end of the stream file is encountered, it is closed.

```

FUNCTION  OPTIONS(*DIRECT)
*****
DEFINE   FIELD(#FILENO) TYPE(*DEC) LENGTH(3) DECIMALS(0) DE
DEFINE   FIELD(#RETNCODE) TYPE(*CHAR) LENGTH(2)
DEFINE   FIELD(#COMMA) TYPE(*CHAR) LENGTH(1)
DEFINE   FIELD(#OPTIONS) TYPE(*CHAR) LENGTH(256) DESC('Optic
CHANGE   FIELD(#OPTIONS) TO('"Read Text LineTerminator=ALL"')
*****

USE      BUILTIN(STM_FILE_OPEN) WITH_ARGS('"/tmp/updphone.txt"' #
IF      COND('#retncode *NE OK')
MESSAGE  MSGTXT('Error occurred on OPEN')
RETURN
ENDIF
*****

***** Read IFS file updphone.txt until end of file.
***** File contains update for employee phone numbers.
***** Each line of data contains EMPNO,PHONENO with
***** a line terminator of Carriage return line feed.
***** eg A0001,754310
*****      A1007,325 187
*****

DUNTIL  COND('#retncode = EF')
USE     BUILTIN(STM_FILE_READ) WITH_ARGS(#FILENO) TO_GET(
IF     COND('#retncode *EQ ER')
MESSAGE  MSGTXT('Error reading stream file')
RETURN
ENDIF
IF     COND('#retncode *EQ OK')
***** update PSLMST with information from stream file
UPDATE  FIELDS((#PHONEBUS)) IN_FILE(PSLMST) WITH_KEY(#EM
ENDIF
*****

ENDUNTIL
*****

***** Close stream file and finish
USE     BUILTIN(STM_FILE_CLOSE) WITH_ARGS(#FILENO)
MESSAGE  MSGTXT('Phone numbers updated')
RETURN

```

9.229 STM_FILE_READ

⇒ **Note:** Built-In Function Rules.

This Built-In function reads data from the specified stream file. The stream file was previously opened by a STM_FILE_OPEN.

If the stream file was opened with a LineTerminator option, then data up to the specified line terminator will be return by a STM_FILE_READ action.

If the stream file was opened with a LineTerminator=NONE option, then as much data as will fit in the provided Returned Data Blocks will be return by the STM_FILE_READ action.

Related Built-In Functions: STM_FILE_OPEN, STM_FILE_CLOSE, STM_FILE_WRITE, STM_FILE_WRITE_CTL. Refer to [9.228 STM_FILE_OPEN](#) for the Line Terminators used.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	File number (file handle) of the file from which data is to be read. This number was allocated to the stream file by the STM_FILE_OPEN.	1	3	0	0

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Returned data block 1 Data will be returned up to the length of this data block	1	Unlimited		
2	A	Opt	Return Code OK = Data has been read and returned. If a Line Terminator was specified on the open, data up to the nominated terminator would have been returned. If Lineterminator=NONE was specified on the open, data would have been returned in the Data Blocks provided. OV = Data overflow. A Line Terminator was specified on the open and as much data as will fit in the Data Blocks has been returned, however, more data exists to reach the nominated terminator for this line. The subsequent STM_FILE_READ will return the further data from this line up to the nominated terminator. EF = End of file encountered. A read was attempted but no more data is available. The Returned Data Block/s will be returned blank. ER = Error occurred.	2	2		
3	A	Opt	Returned data block 2	1	Unlimited		
4	A	Opt	Returned data block 3	1	Unlimited		
5	A	Opt	Returned data block 4	1	Unlimited		
6	A	Opt	Returned data block 5	1	Unlimited		
7	A	Opt	Returned data block 6	1	Unlimited		

8	A	Opt	Returned data block 7	1	Unlimited		
9	A	Opt	Returned data block 8	1	Unlimited		
10	A	Opt	Returned data block 9	1	Unlimited		
11	A	Opt	Returned data block 10	1	Unlimited		

Example

Refer to [9.228 STM_FILE_OPEN](#).

9.230 STM_FILE_WRITE

⇒ **Note:** Built-In Function Rules.

This Built-In function writes data to the specified stream file. The stream file was previously opened by a STM_FILE_OPEN.

If the stream file was opened with a LineTerminator option, then data provided in the Data Blocks will be written and terminated with the provided terminator.

If the stream file was opened with a LineTerminator=NONE option, then data provided in the Data Blocks will be written to the data stream. A line terminator may be written at the appropriate position in the data stream by use of the STM_FILE_WRITE_CTL Built-In Function.

Related Built-In Functions: STM_FILE_OPEN, STM_FILE_READ, STM_FILE_CLOSE, STM_FILE_WRITE_CTL. Refer to [9.228 STM_FILE_OPEN](#) for the Line Terminators used.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	File number (file handle) of the file to which data is to be written. This number was allocated to a stream file and returned on the STM_FILE_OPEN.	1	3	0	0
2	A	Req	Data Block 1	1	Unlimited		
3	A	Opt	Data Block 2	1	Unlimited		

4	A	Opt	Data Block 3	1	Unlimited		
5	A	Opt	Data Block 4	1	Unlimited		
6	A	Opt	Data Block 5	1	Unlimited		
7	A	Opt	Data Block 6	1	Unlimited		
8	A	Opt	Data Block 7	1	Unlimited		
9	A	Opt	Data Block 8	1	Unlimited		
10	A	Opt	Data Block 9	1	Unlimited		
11	A	Opt	Data Block 10	1	Unlimited		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Return Code OK = Data has been written ER = Error occurred.	2	2		

Example

In this IBM i example, a stream file is written to the IBM i IFS.

The file is written to directory /tmp and named example1.txt

The file is written as text and each line is terminated with a Carriage Return Line Feed. The fields in each line are not trimmed of trailing blanks, consequently the data will appear as fixed format.

```
FUNCTION  OPTIONS(*DIRECT)
```

```
*****
```

```
DEFINE   FIELD(#FILENO) TYPE(*DEC) LENGTH(3) DECIMALS(0) DE
```

```
DEFINE   FIELD(#RETNCODE) TYPE(*CHAR) LENGTH(2)
```

```

DEFINE FIELD(#OPTIONS) TYPE(*CHAR) LENGTH(256) DESC('Options')
CHANGE FIELD(#OPTIONS) TO("WRITE notrim Text lineTerminator= C
*****

USE BUILTIN(STM_FILE_OPEN) WITH_ARGS("/tmp/example1.txt" #
IF COND('#retncode *NE OK')
MESSAGE MSGTXT('Error occurred on OPEN')
RETURN
ENDIF
*****

***** Read file PSLMST and write to the IFS file example1.txt
***** Each line written contains employee name and phone number.
***** Each line written contains employee name and phone number.
*****

SELECT FIELDS((#SURNAME) (#GIVENAME) (#PHONEBUS)) FROM
USE BUILTIN(STM_FILE_WRITE) WITH_ARGS(#FILENO #SURNAME)
IF COND('#retncode *NE OK')
MESSAGE MSGTXT('Error writing to stream file')
RETURN
ENDIF
ENDSELECT
*****

***** Close stream file and finish
*****

USE BUILTIN(STM_FILE_CLOSE) WITH_ARGS(#FILENO)
RETURN

```

9.231 STM_FILE_WRITE_CTL

⇒ **Note:** Built-In Function Rules.

This Built-In function writes Line terminator character/s to the data stream.

Related Built-In Functions: STM_FILE_OPEN, STM_FILE_CLOSE, STM_FILE_WRITE, STM_FILE_READ

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	File number (file handle) of the file to which the line terminator is to be written. This number was allocated to a stream file by the STM_FILE_OPEN.	1	3	0	0
2	A	Opt	Line terminator control character/s to be written. Valid values: CR LF (default value) CRLF NL LFCR	1	10		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
2	A	Opt	Return Code OK = Terminator has been written. ER = Error occurred	2	2		

Example

In this IBM i example, a stream file is written to the IBM i IFS. The file is written to directory /tmp and named example2.txt

The data written contains the employee number from database file PSLMST followed by the skills code for the employee. The skills are separated by a comma. The end of the data for an employee is marked by a Carriage Return Line Feed as in this example:

```
A0001,ADVPGM,CS
```

```
A0008
```

```
A0013
```

```
A0090,CL,COBOL,ECD,INTRO,MANAGE1,MARKET2,MARKET3,MBA,RI
```

A line terminator is not automatically written by the STM_FILE_WRITE because of the open option 'LineTerminator=NONE'. The line terminator is written by STM_FILE_WRITE_CTL at the end of the information for an employee.

```
FUNCTION  OPTIONS(*DIRECT)
```

```
*****
```

```
DEFINE   FIELD(#FILENO) TYPE(*DEC) LENGTH(3) DECIMALS(0) DE
```

```
DEFINE   FIELD(#RETNCODE) TYPE(*CHAR) LENGTH(2)
```

```
DEFINE   FIELD(#COMMA) TYPE(*CHAR) LENGTH(1) DEFAULT(',')
```

```
DEFINE   FIELD(#OPTIONS) TYPE(*CHAR) LENGTH(256) DESC('Optic  
CHANGE   FIELD(#OPTIONS) TO("WRITE Text CodePage=819 LineTerm
```

```
*****
```

```
USE      BUILTIN(STM_FILE_OPEN) WITH_ARGS("/tmp/example2.txt" #
```

```
IF      COND('#retncode *NE OK')
```

```

MESSAGE MSGTXT('Error occurred on OPEN')
RETURN
ENDIF
*****
***** Read file PSLMST and for each employee get his skills
***** Write to stream file employee no and list of skills
***** comma delimited. Terminate with a CRLF.
*****
SELECT  FIELDS((#EMPNO)) FROM_FILE(PSLMST)
USE     BUILTIN(STM_FILE_WRITE) WITH_ARGS(#FILENO #EMPNO)
*****
SELECT  FIELDS((#SKILCODE)) FROM_FILE(PSLSKL) WITH_KEY(#I
USE     BUILTIN(STM_FILE_WRITE) WITH_ARGS(#FILENO #COMM#
ENDSELECT
***** Add line terminator at end of details for employee
USE     BUILTIN(STM_FILE_WRITE_CTL) WITH_ARGS(#FILENO CRI
ENDSELECT
*****
***** Close stream file and finish
*****
USE     BUILTIN(STM_FILE_CLOSE) WITH_ARGS(#FILENO)
RETURN

```

9.232 SYSTEM_COMMAND

⇒ **Note:** Built-In Function Rules.

Executes an operating system command.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Command execution option.</p> <p>X = Execute via synchronous process.</p> <p>B = Execute via synchronous process in silent mode</p> <p>S = Execute via the "system" command interface in a command interpreter window.</p> <p>H = Execute via the "ShellExecute" command interface.</p> <p>A = Execute via asynchronous process.</p> <p>W = Backward compatibility only. Use "S" instead.</p> <p>For IBM i:</p> <p>All options result in synchronous execution of the command.</p>	1	1		
2	A	Opt	Command string part 1	1	256		
3	A	Opt	Command string part 2	1	256		

4	A	Opt	Command string part 3	1	256		
5	A	Opt	Working Directory	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Return/Response Code from the operating system. For IBM i: 0 = command successfully executed 1 = command failed to execute successfully.	7	7	0	0

Technical Notes - all platforms

- By using this Built-In Function you are introducing an operating system dependency into your function.

You, the application builder, are totally responsible for doing this, and there is no guarantee, expressed or implied, that anything you do via this Built-In Function is in any way portable across different operating systems (or even versions of the same operating system).

- Do not use this facility to call the X_RUN entry point. It may be used to "start" or "spawn" another X_RUN process, but not to cause it to be recursively invoked.
- As a general rule on a Windows platform, for the Command execution option, try the "X" option first, and if it does not process the command as you require, try the "S" option.

An overview of the execution options:

- The **"X" option** is recommended as it performs synchronous

execution of the command on all platforms. This causes your LANSAs Function to wait while the command executes. The processes created will be fully under the control of Windows Desktop Heap Management – if it is enabled – otherwise, it will use the parent's desktop heap. Note that a return code of 2 implies that the command cannot be found or a dependent DLL cannot be found. On IBM i and Linux, this option is executed the same as the "S" option.

Following is an "X" option example, to invoke a Windows' notepad.exe to display a text file:

```
USE BUILTIN(SYSTEM_COMMAND) WITH_ARGS('X' 'notepad.exe
```

"X" option does not start a command interpreter so commands like COPY or DEL are not supported. If you need to use these commands, use the "S" option. For a complete list of these commands on a Windows platform, start a Command Prompt and type "Help".

- The **"B" option** is the same as the "X" option, however, the execution is performed in the background and hidden on the Windows platform.
USE BUILTIN(SYSTEM_COMMAND) WITH_ARGS('B' 'c:\temp\x.doc
- The **"S" option** indicates that the command should be executed as a standard C "system" command. This option starts a command interpreter, so commands like COPY and DEL are supported. For example, use the "S" option to execute a command line:
USE BUILTIN(SYSTEM_COMMAND) WITH_ARGS('S' 'help.exe >
- The **"H" option** is only available in the Windows environment and invokes the operating system linked editor/viewer for the file specified in the following arguments. Thus if command string part 1 contained **c:\temp\test.doc** then the operating system editor associated with a .doc file (probably MS Word) would be started. Likewise **c:\temp\test.htm** would probably cause an HTML browser to be started. The editor/viewer is executed asynchronously. For example, the "H" option to display a text file using the system's default application:
USE BUILTIN(SYSTEM_COMMAND) WITH_ARGS('H' 'c:\temp\x.doc
- The **"A" option** is the same as the "X" option, except it performs **asynchronous** execution of the command. It is fully supported on all supported Windows platforms. It has been implemented for other platforms, but it may or may not work as expected and LANSAs

reserves the right to not address such issues.

- The "**W**" **option** exists for backward compatibility. The "S" option should be used for new applications.

Technical Notes - Windows & Linux

- Each operating system has differing rules about commands and about the environments in which they are allowed to be executed.
- It is solely the application builder's responsibility to test that commands are valid, and that they function as expected.
- When an application design has a critical point involving the use of this Built-In Function, make sure it does what you expect it to do before basing an application design around it.

Technical Notes - IBM i

- The operating system command specified must be eligible to be executed via the IBM Execute Command (QCMDEXC) API.
- Commands executed via SYSTEM_COMMAND adopt the authority of the LANSAs system owner user profile and the user profile of any other entries in the call stack with USRPRF(*OWNER) as long as the chain is not broken by an entry in the call stack with USEADPAUT(*NO).

If this does not suit your site security policy, then use this command:

**CHGPGM PGM(M@SYEXEC) USRPRF(<your value>)
USEADPAUT(<your value>)**

You should check these values after any upgrade to your LANSAs system.

9.233 TCONCAT

⇒ **Note:** Built-In Function Rules.

Concatenates up to five alphanumeric strings to form one string as a return value. Trailing blanks from each string are truncated during the concatenation operation.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	1st string to concatenate	1	Unlimited		
2	U	Req	2nd string to concatenate	1	Unlimited		
3	U	Opt	3rd string to concatenate	1	Unlimited		
4	U	Opt	4th string to concatenate	1	Unlimited		
5	U	Opt	5th string to concatenate	1	Unlimited		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	U	Req	Concatenated result string	1	Unlimited		
2	N	Opt	Length of returned string	1	15	0	0

9.234 TEMPLATE_@@ADD_LST

⇒ **Note:** Built-In Function Rules.

Allows a new field to be added to an application template list. The application template list is not cleared by this operation. If the field is already in the list it is replaced, otherwise it is added to the list.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of application template that will be used later via the EXECUTE_TEMPLATE Built-In Function.	1	10		
2	N	Req	Number of list that field is to be added to.	1	2	0	0
3	A	Req	Name of field to be added to application	1	10		

		template list. Must be a valid field in the LANSAs data dictionary.				
--	--	---	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.235 TEMPLATE_@@CANSNNN

⇒ **Note:** Built-In Function Rules.

Allows an application template character reply (@@CANSnnn) variable to be set before executing an application template via the EXECUTE_TEMPLATE Built-In Function.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of application template that will be used later via the EXECUTE_TEMPLATE Built-In Function.	1	10		
2	N	Req	Number of @@CANSnnn variable that is to be set.	1	2	0	0

3	A	Req	Value that @@CANSnnn variable is to be set to.	1	74		
---	---	-----	--	---	----	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.236 TEMPLATE_@@CLR_LST

⇒ **Note:** Built-In Function Rules.

Clears an application template list.

Application template lists are widely used by application templates to control and organize the generation of RDML code. They should not be confused with working lists or browse lists which are RDML level constructs used in normal RDML application.

This Built-In Function allows access to an application template list, thus providing a means by which information can be set up for (and thus communicated to) an application template that will be later executed via the EXECUTE_TEMPLATE Built-In Function to generate RDML code.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec

1	A	Req	Name of application template that will be used later via the EXECUTE_TEMPLATE Built-In Function.	1	10		
2	N	Req	Number of application template list to be cleared.	1	2	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.237 TEMPLATE_@@GET_FILS

⇒ **Note:** Built-In Function Rules.

From a nominated base file name this facility returns to the caller a list of all related files.

Functionally this Built-In Function acts like the first step of the application template command @@GET_FILS in that from a nominated base file name it returns a list of files.

In a template, the command displays the resulting list to the user for selection. However, the Built-In Function version returns to the list the calling RDML function.

In an application template the user selects files by entering a non-blank value beside them at the workstation. To perform this action from an RDML function use the TEMPLATE_@@SET_FILS Built-In Function.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Please Note: This Built-In Function has considerably more power than its online template equivalent @@GET_FILS.

The basic difference is in the ability of this function to extract a much more comprehensive file access route list. The online version prevents the extraction of the same underlying physical file more than once in the complete file list. However this Built-In Functions relaxes this rule so that the same underlying physical file cannot be used more than once in any single access route "chain" or "path" starting from, and including, the base file.

Obviously this limit must be imposed to prevent "closed circuits" or "infinite loops" within the access route "chain" or "path".

It is strongly recommended that any developer who plans to use this function design a simple test function using this Built-In Function to extract and display the resulting file list from a nominated base file. This way the characteristics of this function can be much more easily examined and understood.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order

Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment only](#).

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of application template that will be used later via the EXECUTE_TEMPLATE Built-In Function.	1	10		
2	A	Req	Primary or base file name.	1	10		
3	N	Opt	From file number. Default value is 1.	1	2		
4	N	Opt	To file number. Default value is 50.	1	2		
5	A	Opt	Physical Files Only. Must be Y or N. Default value is Y.	1	1		
6	A	Opt	1:1 Relationships Only. Must be Y or N. Default value is Y.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		
2	L	Req	Returned list of related files First entry will be the base file. Working list must have an aggregate length of 40 bytes and is formatted as follows: From - To Description 1 - 1 Selection Flag. Format Alpha. First entry returned as X. Others returned as blanks. 2 - 11 File Name. Format Alpha. 12 - 21 File Library. Format Alpha. 22 - 22 File Type P = Physical L = Logical. 23 - 32 Underlying Physical File. Format Alpha. If this file is a physical file then this name will be the same as the entry file name. 33 - 35 Related file entry number. Format Signed, length 3, decimal 0. 1 = the file is directly related to the base file or is the base file. other = file is indirectly related to the base file via the file specified by this entry number. 36 - 36 Nature of relationship between this file and the related file. Format Alpha. P = Primary or Base File. O = One to One M = Many. 37 - 40 <<Future expansion.>> Format Alpha.	.			

9.238 TEMPLATE_@@NANSNNN

⇒ **Note:** Built-In Function Rules.

Allows an application template numeric reply (@@NANSnnn) variable to be set before executing an application template via the EXECUTE_TEMPLATE Built-In Function.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of application template that will be used later via the EXECUTE_TEMPLATE Built-In Function.	1	10		
2	N	Req	Number of @@NANSnnn variable that is to be set.	1	2	0	0

3	A	Req	Value that @@NANSnnn variable is to be set to.	1	15		
---	---	-----	--	---	----	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.239 TEMPLATE_@@SET_FILS

⇒ **Note:** Built-In Function Rules.

Allows file(s) from a list of files previously built by the TEMPLATE_@@GET_FILS Built-In Function to be "selected" for use within an application template that will be executed later.

Functionally this Built-In Function acts like the second step of the application template command @@GET_FILS in that a "selection" of files is made and set up for use by the application template.

In an application template the user selects files by entering a non-blank value beside them at the workstation. To perform this action from an RDML function use this Built-In Function.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSA for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec

1	A	Req	Name of application template that will be used later via the EXECUTE_TEMPLATE Built-In Function.	1	10		
2	L	Req	<p>List of related files previously built by the TEMPLATE_@@GET_FILS command with the "selection" flag set to a non-blank value to indicate a file that is to be selected.</p> <p>The first entry in the list is the base file and it must always be selected. Working list must have an aggregate length of 40 bytes and is formatted as follows:</p> <p>From - ToDescription</p> <p>1 - 1 Selection Flag. First entry returned as X. Others returned as blanks.</p> <p>2 - 11 File Name. Format Alpha.</p> <p>12 - 21 File Library. Format Alpha.</p> <p>22 - 22 File Type: P = Physical. L = Logical.</p> <p>23 - 32 Underlying Physical File. If this file is a physical file then this name will be the same as the entry file name. Format Alpha.</p> <p>33 - 35 Related file entry number. Format Signed(3,0). 1 = the file is directly related to the base file or is the base file. other = file is indirectly related to the base file via the file specified by this entry number.</p> <p>36 - 36 Nature of relationship between this file and the related file. P = Primary or Base File. O = One to One M = Many.</p> <p>37 - 40 <<future expansion>> A(4)</p>				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
----	------	-------------	-------------	------------	------------	------------	------------

1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		
---	---	-----	---	---	---	--	--

9.240 TEMPLATE_@@SET_IDX

⇒ **Note:** Built-In Function Rules.

Allows an application template index variable to be set to a nominated value before executing an application template via the EXECUTE_TEMPLATE Built-In Function.

This Built-In Function can only be used against a function that has been previously placed into an edit session by using the START_FUNCTION_EDIT Built-In Function.

Special Note: This Built-In Function provides access to very advanced facilities that basically allow RDML functions to construct new RDML functions.

This is a very specialized area that requires very good knowledge of the LANSAs product. Use of this Built-In Function in a "commercial" application (e.g. Order Entry) is not normal and should not be attempted.

⇒ This is a Specialized Built-In Function for use in a [Development Environment](#) only.

For use with

LANSAs for i	YES	Do not use on IBM i in an RDMLX partition.
Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	NO	

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Name of application template that will be used later via the EXECUTE_TEMPLATE Built-In Function.	1	10		
2	A	Req	Identifier of index variable that is to be set.		2		
3	N	Req	Value that index variable is to be set to.	1	2	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = operation completed ER = fatal error detected	2	2		

9.241 TRANSFORM_FILE

⇒ **Note:** Built-In Function Rules.

Transforms the current contents of a disk file into one or more working lists. It is designed to facilitate the transfer of information between Visual LANSA applications and other products such as spreadsheets.

Note: This Built-In Function is designed to work only with text data files.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Name of the primary working list that is to receive the transformed file data.				
2	A	Req	Name of file containing input data.	1	256		
3	A	Opt	Input File Format. A = Normal Delimited File. B = DBF File. (Not available on IBM i.) C = Column File (with signs) D = Column File (without signs) O = Comma Delimited File. T = Horizontal Tab Delimited File. The default value is 'A'. The following special testing options are also available. These options are for	1	3		

			<p>testing and debugging only and must not be used in production software.</p> <p>CL = Expected Column File (with signs) layout.</p> <p>DL = Expected Column File (without signs) layout.</p> <p>Input File Formats A, C, D, O, T, CL and DL support UTF-8 format. This is indicated by placing a 'U' in the second column of this argument (i.e. Format). For example, UTF-8 input for format A would have an Input File Format of 'AU'. Format CL would have an Input File Format of 'CUL'.</p>				
4	A	Opt	<p>Method of handling invalid characters encountered within alphanumeric fields.</p> <p>B = Replace by blank character.</p> <p>I = Ignore. Include character.</p> <p>R = Remove from input.</p> <p>The default value is B.</p>	1	1		
5	A	Opt	<p>Expect Carriage Returns.</p> <p>N = Do not expect carriage return.</p> <p>T = Process a truncated line as if it was padded with blanks.</p> <p>Y = Expect carriage return.</p> <p>The default is 'Y'.</p>	1	1		
6	A	Opt	<p>Decimal Point to be expected. The allowable values are:</p> <p>R = Decimal points do not exist and should be implied from the definition of the field being loaded. Only valid with file formats C and D.</p> <p>other - The value to be expected as a decimal point character.</p>	1	1		

			The default is the currently defined system decimal point (ie: '.' or ','). Note: The use of European style ',' decimal points may create problems in files formats that also use commas to delimit fields.				
7	A	Opt	Close input file option. Y = Close the file at completion. N = Do not close the file at completion unless the end of the file is encountered. The default is Y.	1	1		
8	A	Opt	Record Selection Option. N = No comparison required. EQ = Equal GT = Greater Than GE = Greater Than or Equal To LT = Less Than LE = Less Than or Equal To NE = Not Equal The default is N. This option is not available when input file type is B.	1	20		
9	N	Opt	Record Selection Position. The position in the input record that is to be compared with the compare value. The first byte in the record has the position 1. The default is 1. This argument is ignored if argument number 8 is specified as N.	1	Unlimited	0	0
10	u	Opt	Record Selection Compare Value. When a record is read from the input buffer it	1	256		

			<p>is compared with this value (at the position specified). The number of bytes to be compared is the length of the field which holds the value.</p> <p>If the comparison is true the record is processed. If it is false the record is skipped. The comparison is done with the current value of this field for its full length in its current case. This argument is ignored if argument number 8 is specified as N.</p> <p>There is no default. If *DEFAULT is specified, no comparison will happen.</p>				
11 - 20	L	Opt	<p>Allows up to 10 "appendage" working lists to be specified. Refer to the following notes for more details.</p> <p>Valid only if the primary working list is an RDML list.</p>				

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return Code.</p> <p>EF - File transformed into list(s). End of input file was encountered before the list overflowed.</p> <p>FE - Format Error. For example, Input file format is specified as UTF-8 but file is inUTF-16 format.</p> <p>OV - File transformed into list(s). End of input file not encountered before list was full (ie: overflow).</p> <p>ER - Error when opening file.</p>	2	2		

Technical Notes - TRANSFORM_FILE

File Name

Refer to TRANSFORM_LIST's [9.242.1 Output File Formats](#) for detailed information.

Input File Format

Refer to TRANSFORM_LIST's [9.242.1 Output File Formats](#) for detailed information.

The special options CL and DL are available for testing and debugging purposes only. Do not use these options in production software. When used, these options do not attempt to read the nominated lists. Instead, the layout details are prepared, and the expected layout of the input file is actually printed back into the file specified in the file parameter. If the nominated file does not exist, it is created. If it does exist, then all existing data in the file is erased before the layout details are printed into it. Once the layout file has been produced it can be reviewed by any standard text editor. This information allows you to compare the actual input data file record layout with what this Built-In Function expects the layout to be. Such a comparison can be used to locate formatting problems.

Invalid Character handling in Alpha, Char, String, BLOB, CLOB Fields

Refer to *Invalid Character handling in Alpha, Char, String, BLOB, CLOB Fields* in TRANSFORM_LIST's [9.242.1 Output File Formats](#) . The same rules apply, but to fields in the working lists rather than to those in the output files.

There are some special characters which cannot be processed properly by this Built-In Function. These are: Carriage Return (binary value in Windows: 0x0D), Line Feed (binary value in Windows: 0x0A) and CTRL+Z (binary value in Windows: 0x1A). For example, TRANSFORM_FILE will stop reading the file when it encounters '0x1A' character in Windows.

The best way to output binary data to a file by TRANSFORM_LIST is to convert it into alphanumeric strings by using the Built-In Function [9.9 BINTOHEX](#), then output the result by TRANSFORM_LIST.

To retrieve the data back, use TRANSFORM_FILE to read the data then use [9.137 HEXTOBIN](#) to convert the data back to the original format.

Decimal Point Character

Normally, this option is only required by customers, in some European countries, who have configured their systems to use a comma (,) as the decimal point delimiter. By default the Built-In Function will expect a comma. This may not be appropriate when using data output by other products that do not produce the comma. This option may be used to force this BIF to expect the full stop/period character (.).

Its other use involves the special R (remove) option that may be used when reading files created by applications that use fixed record formats and "implied (by position)" decimal points. In this case the number of decimal positions is implied by the definition of the receiving field in the working list. Refer to the TRANSFORM_LIST command for more details.

The R option can only be used with file formats C and D.

Close Input File Option

This option prevents this Built-In Function from closing the input file when it has completed execution, unless the end of the file is encountered.

In normal use, a call is made to this Built-In Function, it clears the lists, loads as much data as will fit into the lists, closes the input file and returns control to the invoking function. The return code will be set to "EF" or "OV" indicating whether the entire input file would fit into the list.

However, by using the "do not close" option it is possible to perform more complex processing such as:

Reading input files that have more than 9999 records and/or avoid making huge working lists, which require a large amount of allocated memory.

```
def_list #list fields(...) listcount(#count)
```

```
    type(*working) entrys(100)
```

```
dowhile (#retcode *ne EF)
```

```
    use TRANSFORM_FILE into #list (with "do  
        not close" option)
```

```
    execute processlist
```

endwhile

execute processlist

The above example will read any number of records from the input file, even though the list being used is efficiently sized with 100 entries. The list is acting like an input buffer for the application.

Some tips for using this option, and for using this Built-In Function, are:

- This function is designed to be an interface between Visual LANSA applications and external applications. It is designed to open a file, read data from it, then close it again. It is not designed to service more complex "system" level tasks such as maintaining an "always open" polling file.
- Up to 50 input files may be open concurrently. The operating system you are using may have limitations or configuration options that lower this limit.
- There is no limitation on maximum record length. At the end of every record a New Line character will be added as an End Of Record delimiter.
- When using the "Keep Open" option always let the Built-In Function continue until "EF" is returned. This means that the input file will have been closed.
- This Built-In Function must check all arguments every time it is called, and also search through a list of currently opened input files looking for a match. Therefore it is most efficient when called just a few times with lists allowing many entries, and least efficient when it is called many times with list(s) allowing just a few entries.

Record Selection Capabilities

Not available when the input file type is B.

Arguments 8, 9 and 10 allow simple record selection logic to be performed. If this option is enabled by passing argument 8 as a valid value other than N, then as each <record> is read from the file the following expression is evaluated:

if (substring(<record>,<position>,<length>) <operation> <value>)

where:

- **<record>** is the current record.
- **<position>** is the argument 9 position.
- **<length>** is the length of the argument 10 field. This <length> should not be

bigger than 256 bytes.

- **<operation>** is the argument 8 operation.
- **<value>** is the value of the argument 10 field.

If the expression is found to be true, the record is selected. If the expression is false then the record is ignored.

Note that this comparison happens before any data processing.

It is a byte-orientated operation, not field-orientated operation.

If a file is planned to be used with Record Selection, it is recommended that you use a fixed length columns file (C or D types in TRANSFORM_LIST). It is also recommended that you move all the variable length fields toward the tail of the record in a such way so the <position > appears before those fields. The purpose of doing this is to make sure that the portion of <length> bytes, starting from <position> in the input buffer, has the same meaning for all records. Also if the <position> + <length > is greater than the number of bytes in the record, TRANSFORM_FILE will give this fatal error "Invalid arguments for comparison" and stop.

If the input file type is A, O or T, then any implied character appearing before the data to be compared should be counted in the <position > calculation.

Example:

An A type file is built from a list of 2 Alpha fields. Both are 9 bytes long. Fields of any entry are always full (occupy maximum length). There is no invalid character in the data. The comparison is intended to be from the first byte of the second field. So the <position> must be calculated like this:

$\langle \text{position} \rangle = 1(\text{opening double quote for the first field}) + 9(\text{length of the first field}) + (1 \text{ closing double quote}) + 1(\text{coma}) + 1(\text{opening double quote for the second field}) + 1(\langle \text{position} \rangle \text{ starts from } 1) = 14$

If the output file type is C or D the <position> calculation is:

$\langle \text{position} \rangle = 9(\text{length of the first field}) + 1(\langle \text{position} \rangle \text{ starts from } 1) = 10$

If you need help in understanding why a record is or is not located, turn tracing on to level 9, category 'BIF'. The trace file will list the comparison parameters used and the data that did not match the comparison parameters. Note that the trace is not in Unicode so Unicode data may not be displayed as you see it when using a Unicode-aware program like Notepad.

Appendage Lists

Up to 10 appendage working lists may be specified when invoking this Built-In Function. Appendage lists may be used when the input file contains more than 100 fields or where the aggregate entry length of a list exceeds 256 bytes.

Refer to the [9.242 TRANSFORM_LIST](#) Built-In Function for more details of appendage lists. The concept of the "appendage option" field has no meaning to this Built-In Function. All fields defined in appendage list(s) are processed just like a logical extension of the primary list (argument 1). Appendage working lists should all have the same maximum number of entries allowed as the primary list.

Error Handling and Error Activity

The following table indicates the types of errors that you can trap at the RDML level with an "ER" return code (User Trap) and those that will cause a complete failure of your application (System Error). System errors invoke Visual LANSA full error handling and cause the entire X_RUN "session" to end. They cannot normally be trapped at the RDML level.

Type Of Error	Resulting Action
Attempt to open too many input files	System Error
Input file option is not A, T, C, D, B or O	System Error
Invalid character option is not I, B or R	System Error
Carriage control option is not Y, N or T	System Error
Close file option is not Y or N	System Error
Appendage list has wrong maximum entries value	System Error
Error when attempting to open input file	User Trap
Error while reading from input file	System Error
Bad or unexpected data in input file	System Error

Error Handling Note

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of

your application.

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
endif
```

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

Special note

BLOB, CLOB fields values in Results Lists

BLOB (or CLOB) field holds only the file name. If the value of File Name is <drive>:\<path>\<file>.<suffix> then the BLOB (or CLOB) file itself is located under the subdirectory: <drive>:\<path>\<file>_LOB\ .

For example:

To get a BLOB value BLOBNumber1.txt from a transformed file:

C:\Root\Data\Transformed1.dat

then the BLOB file itself must be:

C:\Root\Data\Transformed1_LOB\BLOBNumber1.txt

For a transformed file created NOT by TRANSFORM_LIST, BLOB (or CLOB) files need NOT to be duplicated. In this case the BLOB (or CLOB) value must have the full path in it.

For example, if you have a BLOB file called BLOBNumber1.txt currently located in C:\Data\ and if you do not want to duplicate the BLOB file, the value of the BLOB field in the transformed file must be C:\Data\BLOBNumber1.txt.

Example

Refer to [9.242 TRANSFORM_LIST](#).

9.242 TRANSFORM_LIST

⇒ **Note:** Built-In Function Rules.

Transforms the current contents of one or more working lists into a disk file. It is designed to facilitate the transfer of information between Visual LANSA applications and other products (e.g: spreadsheets).

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	L	Req	Name of the primary working list that is to be transformed into a disk file. Note: If this list contains fields of type Binary or VarBinary, the Built-In Function will end in error.				
2	A	Req	Name of file to be replaced or created by this Built-In Function.	1	256		
3	A	Opt	Output File Format. A - Normal Delimited File. B - DBF File. (Not available on IBM i.) C - Columnized File (Numeric Fields Have Leading Signs) with signs. D - Columnized File (Numeric Fields do NOT have Signs) without signs.	1	3		

			<p>O - Comma Delimited File.</p> <p>S – Comma Delimited File: A common CSV file format. It has exactly the same format as type O files except that a completely blank field is not represented as a single blank and trailing blanks are ONLY included if they represent invalid character substitutions.</p> <p>T - Horizontal Tab Delimited Files.</p> <p>Output File Formats A, C, D, O, S and T support UTF-8 format. This is indicated by appending a 'U' to this argument (i.e.Format). For example, UTF-8 output for format A would have an Output File Format of 'AU'. The default value is 'A'.</p>				
4	A	Opt	<p>Method of handling invalid characters encountered within alphanumeric fields.</p> <p>'B' - Replace by blank character.</p> <p>'I' - Ignore. Include character.</p> <p>'R' - Remove from output.</p> <p>The default value is 'B'.</p>	1	1		
5	A	Opt	<p>Include Carriage Return at the end of each record.</p> <p>'N'- Do not include carriage return.</p> <p>'T' - Include carriage return and also truncate all blank data from the end of the record.</p> <p>'Y'- Include carriage return.</p> <p>The default is 'Y'.</p>	1	1		
6	A	Opt	<p>Decimal Point to be used . The allowable values are:</p> <p>'R'- Remove the decimal point from all numeric representations. This will shorten the length of numeric fields that have decimal positions by 1 character. Only valid with file formats C and D.</p>	1	1		

			<p>other - The value to be used as a decimal point character.</p> <p>The default is the currently defined system decimal point (i.e. '.' or ',').</p> <p>Note: The use of European style ',' decimal points may create problems in files formats that also use commas to delimit fields.</p>				
7	A	Opt	<p>Close Output File Option.</p> <p>'Y'- Close the file at completion.</p> <p>'N'- Do not close the file at completion.</p> <p>The default is 'Y'.</p>	1	1		
8 - 17	L	Opt	<p>Allows up to 10 Appendage Working Lists to be specified. Refer to the following notes for more details.</p> <p>Valid only if the primary working list is an RDML list.</p>				

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return Code.</p> <p>OK - File Created.</p> <p>ER - Error when opening file. Refer to Return Code - Error Handling and Error Activity .</p>	2	2		

Technical Notes - TRANSFORM_LIST

[9.242.1 Output File Formats](#)

[9.242.2 Other Parameters](#)

SQLNULL Handling

When a field is SQLNULL, the *NULL equivalent is output.

Special Handling for BLOB and CLOB value

The full BLOB/CLOB file name will be saved in the output file. The BLOB/CLOB file itself will be duplicated in a subdirectory under the output file directory. The name of this subdirectory is <output file name>_LOB. For example if the output file is:

C:\Root\Data\Transformed1.dat,
and the original CLOB file is C:\XYZ\CLOBNumber1.txt
then the duplicated CLOB file is

C:\Root\Data\Transformed1_LOB\CLOBNumber1.txt
and the CLOB value in the Transformed1.dat will be:
C:\XYZ\CLOBNumber1.txt

If Transformed1.dat is moved (or copied) into another system, move or copy the sub directory Transformed1_LOB and all its contents as well.

Example

The following outline function can save the contents of an existing SQL table to a disk file, or insert the contents of disk file into an SQL table (i.e. An SQL table Save/Restore function)

```
def_list #list fields(...) listcount(#count) type(*working) entrys(100)
request fields (#option and name of disk file involved)
if (#option = SAVE)
  select fields(...) from_file(...)
    add_entry #list
    if (#count = 100)
      use TRANSFORM_LIST #list (with "do not close" option)
      clr_list #list
    endif
  endselect
  use TRANSFORM_LIST #list (with "close" option)
else (#option was RESTORE)
  dowhile (#retcode *ne EF)
    use TRANSFORM_FILE into #list (with "do not close" option)
    execute insertlist
  endwhile
  execute insertlist
```

```

endif
subroutine insertlist
  selectlist #list
    insert fields(...) to_file(...)
  endselect
  clr_list #list
endroutine

```

By adding the CONNECT_SERVER and CONNECT_FILE Built-In Functions, this function could be very simply expanded to support the following table of "data transfers":

Data Target	<----- Data Source ----->		
	PC SQL Table	PC Disk File	IBM i File
PC SQL Table	N/A	Yes	Yes
PC Disk File	Yes	N/A	Yes
IBM i File	Yes	Yes	N/A

9.242.1 Output File Formats

Currently the following output formats are supported. In the examples this notation is used:

<OptCR> Indicates an optional carriage return character.

<nl> Indicates a new line (or line feed) character.

<t> Indicates a horizontal tab character.

[A - Normal Delimited File](#)

[B - DBF File](#)

[C - Columnized File \(Numeric Fields Have Leading Signs\)](#)

[D - Columnized File \(Numeric Fields do NOT have Signs\)](#)

[O - Comma Delimited File](#)

[T - Horizontal Tab Delimited Files](#)

[dBASE III PLUS Field Format Supported](#)

A - Normal Delimited File

One of the most common file formats. Alpha, BLOB, CLOB fields are enclosed in double quote signs (") and trailing blanks are removed. A completely blank field appears as a single blank enclosed in double quotes (i.e. " ").

Date, DateTime, Time, Char and String fields are saved like Alpha. In Char and String fields trailing blanks are not removed.

Integer, Float, Packed and Signed fields appear with a leading negative sign, leading zeros suppressed and a decimal point (where required).

Boolean field occupies 1 byte. FALSE value appears as 'F'. TRUE appears as 'T'.

Individual fields are delimited by a comma (,).

The length of each file record may vary.

The position of an individual field within a record may vary from record to record. This is not a "fixed format" file. For example:

```
"SMITH","IAN",2153,345.56,"ADM",-456.78<OptCR><nl>
```

B - DBF File

This option tells the BIF to produce dBASE III PLUS, without memo DBF files. It is another common file format used to exchange information with other environments such as Microsoft Excel. Refer to [dBASE III PLUS Field Format Supported](#) more details.

There are limitations to this option. Below are listed both supported and non-supported types.

Supported Types

Alpha/Char/String (up to 254 bytes only), stored as Character

Packed and Signed field values are stored as DBF Numeric.

Date field value is stored as DBF Date

Date/Time field value is stored as DBF Timestamp. Fraction may be lost during the transforming process.

A Timestamp value occupies 8 bytes in DBF file – two longs, first for date, second for time. The date is the number of days since 01/01/4713 BC (Julian day) . Time is hours * 3600000L + minutes * 60000L + Seconds * 1000L. Please refer [Julian Day Count](#) for details about Julian Day Count algorithm.

Time field value is stored as DBF Timestamp. But only last 2 bytes will be used. The value of the first 2 bytes will be 1900-01-01

Integer field value is stored as DBF Numeric. 1 byte Integer occupies 4 bytes including sign. 2 bytes Integer occupies 6 bytes including sign . 4 bytes Integer occupies 11 bytes including sign. 8 bytes Integer occupies 20 bytes. The value firstly is converted to a string (radix 10), if the length of the string is smaller than the required length, it will be right padded by BLANKs.

Float field value is stored as DBF Double.

Boolean field value is stored as DBF Logical

Unsupported Types

Alpha, Char, String with length > 254, Integer 8, BLOB and CLOB.

C - Columnized File (Numeric Fields Have Leading Signs)

Produces a columnized or "fixed format" file. The length of each file record is identical (unless appendage lists are used ... see later note) and the position of an individual field is identical within each record.

The width of a field's resulting column (and thus the overall "record layout" of the file) can be predicted by the following rules:

- Alpha, BLOB, CLOB Date, DateTime, Time , Char and String fields have the same length as their definition in the Data Dictionary or RDML function.
- For Packed and Signed fields the length is their Total Digits defined in the Data Dictionary or RDML function plus 1(for the leading sign). Note that this is the "Total Digits", not the computed storage or byte length for Packed fields.
- 8 bytes Integer field occupies 20 bytes in the output file (include the leading sign).
- 4 bytes Integer field occupies 11 bytes in the output file (include the leading sign).
- 2 bytes Integer field occupies 6 bytes in the output file (include the leading sign).
- 1 byte Integer field occupies 4 bytes (include the leading sign).
- 4 bytes Float field occupies 14 bytes (include the leading sign)
- 8 bytes Float field occupies 23 bytes(include the leading sign)
- Unlike type A and T formats, which do not show a leading positive sign, this format always includes a leading positive or negative sign. Even UNSIGNED Integer will have a positive sign.

If the field has decimal positions (and the option to remove decimal points is not being used) increment the length by 1 to allow for the decimal point(this is not applicable for Float, where the dot '.' presence is compulsory). For example:

A(12) A(9) S(4,0) P(9,2) A(3) P(7,2)

.....:.....:.....:.....:..... SMITH IAN +2153+0000345.56ADM-
00456.78<OptCR><nl>

D - Columnized File (Numeric Fields do NOT have Signs)

- Produces a file exactly as for type "C" except that Packed, Signed, Integer and Float fields do not have a leading sign. For examples:

With decimal points included:

A(12) A(9) S(4,0)P(9,2) A(3) P(7,2)

.....:.....:.....:.....:..... SMITH IAN 21530000345.56ADM00456.78<OptCl
<nl>

With decimal points removed:

A(12) A(9) S(4,0)P(9,2) A(3) P(7,2)

.....:.....:.....:.....:..... SMITH IAN 2153000034556ADM0045678<OptCR>
<nl>

- All the rules applied to predict the width of a fields resulting column are the same as for type "C" with an exception for Packed, Signed, Integer or Float fields: 1 must be subtracted from the result.

Float field Note:

The Float field value is converted into "scientific notation" representation.

For example:

The biggest double float value that can be handled by some C compilers in "scientific notation" representation is: 1.7976931348623158E+308

O - Comma Delimited File

A common file format. It has exactly the same format as type A files except that Alpha, BLOB, CLOB Date, DateTime, Time , Char and String fields are not enclosed in double quotes.

T - Horizontal Tab Delimited Files

A popular file format. It has exactly the same format as type A files except that Alpha, BLOB, CLOB Date, DateTime, Time , Char and String fields are not enclosed in double quotes and individual fields are delimited by the horizontal tab character. For example:

SMITH<t>IAN<t>2153<t>345.56<t>ADM<t>-456.78<OptCR><nl>

dBASE III PLUS Field Format Supported

Symbol	Data Type	Description
@	Timestamp	8 bytes - two longs, first for date, second for time. The date is the number of days since 01/01/4713 BC. Time is hours * 3600000L + minutes * 60000L + Seconds * 1000L
+	Autoincrement	Same as a Long
B	Binary, a string	10 digits representing a .DBT block number. The number is stored as a string, right justified and padded with blanks.
C	Character	All OEM code page characters - padded with blanks to the width of the field.
D	Date	8 bytes - date stored as a string in the format YYYYMMDD.
F	Float	Number stored as a string, right justified, and padded with blanks to the width of the field.
G	OLE	10 digits (bytes) representing a .DBT block number. The number is stored as a string, right justified and padded with blanks.
I	Long	4 bytes. Leftmost bit used to indicate sign, 0 negative.
L	Logical	1 byte - initialized to 0x20 (space) otherwise T or F.
M	Memo, a string	10 digits (bytes) representing a .DBT block number. The number is stored as a string, right justified and padded with blanks.
N	Numeric	Number stored as a string, right justified, and padded with blanks to the width of the field.
O	Double	8 bytes - no conversions, stored as a double.

9.242.2 Other Parameters

[Appendage Working Lists](#)

[Close Output File Option](#)

[Decimal Point to be used](#)

[Return Code - Error Handling and Error Activity](#)

[File Name](#)

[Invalid Character handling in Alpha, Char, String, BLOB, CLOB Fields](#)

[Julian Day Count](#)

File Name

Must be formatted correctly for the operating system being used. For Windows either a fully qualified name in the format <drive>:\<path>\<file>.<suffix> may be used, or a shortened form such as <file>.<suffix>. The shortened form will replace or create the file in the current directory.

The <suffix> value DAT is conventionally used for permanent data files and TMP for temporary files. If the file exists it is opened and all existing data is erased. If the file does not exist it is created and then opened.

Invalid Character handling in Alpha, Char, String, BLOB, CLOB Fields

Some character values may corrupt the output file if they are inserted into the output data. This option specifies what should happen if an invalid character is encountered. The set of invalid characters that are scanned for varies by requested output format as follows:

Format	Invalid Characters
A & B	Horizontal Tab, Vertical Tab, Carriage Return, Form Feed, Back Space, New Line (Line Feed), Double Quotes, End of String Delimiter (i.e. X'00').
O	Horizontal Tab, Vertical Tab, Carriage Return, Form Feed, Back Space, New Line (Line Feed), Comma, End of String Delimiter (i.e. X'00').
T, C & D	Horizontal Tab, Vertical Tab, Carriage Return, Form Feed, Back Space, New Line (Line Feed), End of String Delimiter (i.e. X'00').

Note: Only Alpha, Char, String, BLOB, CLOB fields are scanned for invalid characters.

The supported invalid character handling options are:

- B - Replace by Blank
The character is replaced by a blank character in the output stream.
- I - Ignore
The presence of the character is ignored. It is included into the output stream and may corrupt further processing of the file by other applications.
- R - Remove from Output
The character is removed from the output stream. This option effectively shortens the output field length by 1. You should not use this option when making "fixed format" output files.

Decimal Point to be used

This option is only required when the system is configured to use a comma (,) as the decimal point delimiter. By default, output numeric fields will use the comma but this may not be appropriate when other products will not accept the comma. This option may be used to force this Built-In Function to use of the full stop/period character (.).

The other use for this parameter is the special 'R' (Remove) option that may be used when creating files for input to applications that use fixed record formats and "implied (by position)" decimal points.

The 'R' option can only be used with file formats 'C' and 'D'.

Close Output File Option

The Close Output File option prevents the Built-In Function from closing the output file when it has completed execution.

In normal use, a working list is loaded with data and passed to this Built-In Function. The list is read, written to the disk file, and then the disk file is closed. Subsequent use of this Built-In Function with the same file name will replace the existing file (and its data) with a new set of data.

By using the "do not close" option, much more complex processing may be performed such as in the following example.

To avoid using huge working lists which require a large amount of allocated memory, the following code will create any number of records in the output file even though the list being used is efficiently sized with just 100 entries. The list

is acting like an output buffer for the application.

```
def_list #list fields(...) listcount(#count)
    type(*working) entrys(100)
select fields(...) from_file(...)
    add_entry #list
    if (#count = 100)
        use TRANSFORM_LIST #list (with "do not
            close" option)
        clr_list #list
    endif
endselect
use TRANSFORM_LIST #list (with "close" option)
```

- To produce output files that have mixed record types. Consider an output file containing order details that has two different "record types". One for the order "header" and one for each "detail" item. A function to do this might be structured like this:

```
def_list #head fields(...) type(*working) entrys(1)
def_list #line fields(...) listcount(#count) type(*working) entrys(100)

select fields(...) from_file(orders)
    inz_list #head num_entrys(1)
    use TRANSFORM_LIST #head (with "do not close" option)
    select fields(...) from_file(lines) with_key(...)
        add_entry #line
        if (#count = 100)
            use TRANSFORM_LIST #line (with "do not
                close" option)
            clr_list #line
        endif
    endselect
    use TRANSFORM_LIST #line (with "do not close"
        option)
    clr_list #line
endselect
use TRANSFORM_LIST #line (an empty list with "close"
```

option).

Some tips for using this option, and for using this Built-In Function are:

- This function is designed to be an interface between Visual LANSA applications and external applications. It is designed to open a file, write data to it, then, close it again. It is not designed to service more complex "system" level tasks such as maintaining an "always open" log file.
- Up to 50 output files may be open concurrently. The operating system you are using may have limitations or configuration options that lower this limit.
- There is no limitation on maximum record length. At the end of every record a New Line character will be added as an End Of Record delimiter.
- If the file created is to be read by TRANSFORM_FILE in version 10.0 of LANSA or prior, then the maximum record length must be 20000 bytes.
- Always place a final call to this Built-In Function to cause it to close the output file. Pass the working list as an empty or cleared list if you just want to close the file and not add any more data to it.
- This Built-In Function must check all arguments every time it is called, and also search through a list of currently opened output files looking for a match. Therefore it is most efficient when called just a few times with list(s) containing many entries, and least efficient when it is called many times with list(s) containing just a few entries.

Appendage Working Lists

With the introduction of RDMLX working list, which may contain up to 1000 fields, 2G (2147483647) entries with an entry length of up to 2Gb (2147483647 bytes), in most of the cases you will not need the Appendage Lists. However if RDMLX appendage lists are used, the same rules as for RDML lists are applied (apart from the above). Note that when the primary working list is an RDMLX list, an appendage list cannot be used.

Up to 10 appendage working lists may be specified when invoking this Built-In Function. This Built-In Function uses a driving loop that can be represented like this:

-> do for each entry in the "primary" working list (ie: argument 1)

```
|  
|   map details of primary list entry "n" to output buffer
```

```

|
| -> do for each appendage working list that has been specified.
| |   get a matching entry "n" from the appendage list.
| |   -- if entry "n" can be found
| | |   -- test the case of the entry's "appendage option"
| | | |----> when = A, Append this entry to the current record.
| | | |----> when = N, Write the existing buffer, ends the current record with a
New Line
| | | |   character . Clear the output buffer and map this entry into it to start a
new | | | |   record.
| | | |----> when = O, Omit (ie: skip) this list entry.
| | | |----> otherwise: Issue a fatal error and kill the function.
| | |   -- endcase
| |   -- endif
|   -- enddo
|
|   if the buffer is not empty, write output record from buffer.
|
-- enddo

```

Note the "appendage option". Any working list that is used as an appendage list must have an alphanumeric 1 field as its first defined field. This alphanumeric 1 field is the "appendage option" and indicates how the entry should be handled. The currently supported values are:

A Append this entry to the output buffer being built.

N Write the existing buffer and make a new one with this entry.

O Omit this entry from the output stream.

The appendage option field is not really part of the list. Its value is tested, but it is not output to the output buffer, so it has no bearing in record layout and/or length calculations even though it is actually part of the working list definition.

So far the use of "appendage lists" may not be apparent. However, by using appendage lists you can solve the following problem:

- An RDML working list can contain at most 100 fields. However you may wish to output a file containing 120 fields (say). You can do this by using a primary RDML list of 100 fields and a single appendage list of 21 fields

(including the appendage option field). The following example allows an output file with a record containing up to 199 fields to be created. Since up to 10 appendage lists can be specified this allows a theoretical total of $100 + (10 * 99) = 1090$ fields in an output file record. Similarly some functions have their maximum RDML working list entry length restricted to 256 bytes (this is an IBM i limitation but it is imposed in Visual LANSA to maintain application portability). This limitation can also be easily overcome by using one or more appendage lists.

- If an RDMLX working list with 120 fields is used, the "appendage list" is not required.

The following example covers the "A" (append) option. However the "N" (new) and "O" (omit) appendage options can be effectively used to create "variable record files" in an efficient manner.

```
def_list #plist fields(...) type(*working)
def_list #alist1 fields(#aoption ..... ) type(*working)
change #aoption 'A'
select fields(...) from_file(...)
  fetch fields(...) from_file(...)
  fetch fields(...) from_file(...)
  fetch fields(...) from_file(...)
  add_entry #plist
  add_entry #alist1
endselect
use TRANSFORM_LIST #plist (with appendage list #alist1)
```

Consider some sort of "transaction" output file that you must create for input to an existing mainframe application system.

Every customer must have a type "HDR" header record. There may be a type "ADR" address update record and there may be a type "BIL" billing record. Very old fashioned, but still very common.

By defining a primary list for the HDR data and two appendage lists for the ADR and BIL data (with appendage option fields) a "stream" of mixed format and optional records like this can be created by just one invocation of this Built-In Function. In this case the appendage options O (omit) and N (new) would be used.

HDR-HDR-ADR-HDR-BIL-HDR-ADR-BIL-HDR-HDR-ADR-BIL-HDR

If however, you wanted to create multiple BIL records for a single HDR record you would have to use the "do not close" option. See the reference in the Close Output File Option for more details of how this could be achieved.

Return Code - Error Handling and Error Activity

The following table indicates the types of errors that you can trap at the RDML level with an "ER" return code (User Trap) and those that will cause a complete failure of your application (System Error). System errors invoke Visual LANSA full error handling and cause the entire X_RUN "session" to end. They cannot normally be trapped at the RDML level.

Type Of Error	Resulting Action
Attempt to open too many output files	System Error
Output file option is not A, T, C, D, B or O	System Error
Invalid character option is not I, B or R	System Error
Carriage control option is not Y, N or T	System Error
Close file option is not Y or N	System Error
Invalid appendage option in appendage list	System Error
Error when attempting to open output file	User Trap
Error while writing to opened output file	System Error

Note:

It is very strongly recommended that you avoid building complex error handling schemes into your applications. Use a very simple trap like this at all levels of your application.

```
if (#retcode *ne OK)
    abort msgtxt('Failed to .....')
```

endif

Let the standard error handling Built-In Function to every generated application take care of the problem. Situations have arisen where user defined error handling logic has become so complex as to consume 40 - 50% of all RDML code (with no obvious benefit to the application). Do not fall into this trap.

Julian Day Count

The Julian Day Count is a uniform count of days from the past (-4712 January 1, 12 hours UTC (Universal Coordinated Time - the modern equivalent of Greenwich Mean Time) (Julian proleptic Calendar) = 4713 BCE January 1, 12 hours GMT (Julian proleptic Calendar) = 4714 BCE November 24, 12 hours GMT (Gregorian proleptic Calendar)). At this point, the Julian Day Number is 0.

The Julian Day Count is not related to the Julian Calendar introduced by Julius Caesar.

There are several algorithms of calculating the Julian Day Number. Although they are very similar to each other, the results may be different. The algorithm used by this Built-In Function calculates the Julian Day Number of any date given on the Gregorian Calendar. The Julian Day Number calculated will be for 0 hours, GMT, on that date.

1. Express the date as Y M D, where Y is the year, M is the month number (Jan = 1, Feb = 2, etc.), and D is the day in the month.
2. If the month is January or February, subtract 1 from the year to get a new Y, and add 12 to the month to get a new M. (Thus, we are thinking of January and February as being the 13th and 14th month of the previous year).
3. Dropping the fractional part of all results of all multiplications and divisions, let

$$A = Y/100$$

$$B = A/4$$

$$C = 2-A+B$$

$$E = 365.25x(Y+4716)$$

$$F = 30.6001x(M+1)$$

$$JD= C+D+E+F-1524$$

This is the Julian Day Number for the beginning of the date in question at 0

hours, UTC.

The following calculation is used to convert a Julian Day Number to a Gregorian date, assuming that it is for 0 hours, UTC. Drop the fractional part of all multiplications and divisions.

Note: This method will not give dates accurately on the Gregorian Proleptic Calendar, that is, the calendar you get by extending the Gregorian calendar backwards to years earlier than 1582 using the Gregorian leap year rules. In particular, this method fails if $Y < 400$.

$$Z = JD$$

$$W = (Z - 1867216.25)/36524.25$$

$$X = W/4$$

$$A = Z + 1 + W - X$$

$$B = A + 1524$$

$$C = (B - 122.1)/365.25$$

$$D = 365.25 \times C$$

$$E = (B - D)/30.6001$$

$$F = 30.6001 \times E$$

$$\text{Day of month} = B - D - F$$

$$\text{Month} = E - 1 \text{ or } E - 13 \text{ (must get number less than or equal to 12)}$$

$$\text{Year} = C - 4715 \text{ (if Month is January or February) or } C - 4716 \text{ (otherwise).}$$

Also see

[General Variables](#)

9.243 UNLOAD_FILE_DATA

⇒ **Note:** Built-In Function Rules.

Will call the OAM for the requested file and unload all its data to the flat file specified. If the flat file specified already exists it will be overwritten. See [Note](#) following re version upgrade issues.

This Built-In Function expects to be executed on the same machine as the OAM. Both the BIF and the OAM need to access the output file. If you execute the BIF from a local Function but redirect the File to SuperServer, it is your responsibility to ensure that the output filename is valid on both the client and the server.

For use with

LANSA for i	YES	The unloaded data can be used on other platforms, for example, load the data on Windows using the LOAD_FILE_DATA Built In Function.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No.	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	The LANSAs File Name	1	10		
2	A	Opt	Output File Path/Name Default Value =\X_ppp\source\ <file </file Name>.dat	1	256		

3	A	Opt	Y/N Check for OAM Default = N	1	1		
4	A	Opt	Unload BLOB and CLOB fields. Default = N , meaning BLOB and CLOB are unloaded as their default (*SQLNULL). If Y , if BLOB or CLOB fields exist on the file and are not Null, they are unloaded to files in the same directory as the output file (Arg 2), with the naming convention FileName_Field_DiskFile.ext (where DiskFile.ext is the disk file name saved in the table for the LOB).	1	1		
5	A	Opt	Perform a Commit to release any database locks that the database may leave as part of this operation. Default = N , for backward compatibility. Y will release any database table locks. Sybase leaves a database table locked at the end of this operation. This blocks dropping the table. Set this value to Y to release the database table lock so that the table can be dropped, for example, by REBUILD_FILE.	1	1		
6	A	Opt	CTD Location Level A= All (Partition + System).	1	1		

			P = Partition Level only. S=System Level only. Default is A.				
--	--	--	--	--	--	--	--

Return Values

No.	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code: OK = File successfully unloaded ER = File unload failed (possible causes - invalid path, out of space on disk drive). NT = No table exists NO = No OAM. File does not exist or is not compiled. NO only returned when input option 3 is set to Y.	2	2		
2	L	Opt	Working list of files created for unloaded BLOB and CLOB files	256	256		

Note

As from V10.0, UNLOAD_FILE_DATA will make a copy of the .CTD (Common Table Definition) file with a .CTX (ex-Common Table Data) extension.

The .CTX file must always be with its .DAT file and OAM.

9.244 UNLOCK_OBJECT

⇒ **Note:** Built-In Function Rules.

Releases the lock on the specified User Object

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Object Type.	1	20		
2	A	Opt	Object Identifier 1. ALL = All Identifiers	1	10		
3	A	Opt	Object Identifier 2. ALL = All Identifiers	1	10		
4	A	Opt	Object Identifier 3. ALL = All Identifiers	1	10		
5	A	Opt	Object Identifier 4. ALL = All Identifiers	1	10		
6	A	Opt	Locking Level ANY = Any (Default) FUNC = Function JOB= Job PERM = Permanent	3	4		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return code OK = Object was successfully unlocked. ER = Object could not be unlocked	2	2		

There are some rules concerning the ability and access to locked User Objects. These rules are determined by the Locking Level used to lock the User Object, and are as follows:-

- When a User Object is locked with a 'FUNC' Locking Level it may only be unlocked by the same function and job that the lock was allocated in. If the 'ALL' literal is used for an Object Identifier and multiple User Object locks are found, then only locks allocated in the current function and job are unlocked.
- When a User Object is locked with a 'JOB' Locking Level it may only be unlocked by the same job that the lock was allocated in. If the 'ALL' literal is used for an Object Identifier and multiple User Object locks are found, then only those allocated in the current job are unlocked.
- A User Object locked with a Locking Level of 'PERM' may be unlocked by any job or function.
- User Object locks are also automatically unlocked as determined by the Locking Level specified on the LOCK_OBJECT.
 - A Locking Level of 'FUNC' indicates that the lock will be automatically removed at the end of the function that created it.
 - A Locking Level of 'JOB' indicates that the lock will be automatically removed when you exit LANSAP.
 - A Locking Level of 'PERM' indicates that the User Object lock exists until removed with UNLOCK_OBJECT.

Note: Because some User Object locks are automatically removed, you may not need to use the UNLOCK_OBJECT Built-In Function. By locking the User Object with the appropriate Locking Level you can allow the locks to be automatically released at the end of the function/LANSA.

For further information and examples concerning User Objects and their locking refer to the Built-In Function [9.152 LOCK_OBJECT](#)

9.245 UPDATE_IN_SPACE

⇒ **Note:** Built-In Function Rules.

Updates a single cell row that matches the key values supplied.

For use with

LANSA for i	YES	Only available for RDMLX.
Visual LANSA for Windows	YES	
Visual LANSA for Linux	YES	

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	R	Space Name	1	256		
2- 20	w	O	Fields that specify the values to be used to locate and update the cells row.	1	Unlimited	0	Unlimited

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	O	Standard Return Code "OK" = A cell row was found and updated. "NR" = No cell row could be found to update. "ER" = Update attempt failed. Messages issued will indicate more about the cause of the failure.	2	2		

Technical Notes

You cannot change the values of keys cells in a space object. To alter the values of key cells you must delete and (re)insert the cell row.

The field values must be specified in the same order as the cells in the space were defined. Cells are matched by the order of their specification in arguments 2 -> 20. The names of the fields used have no bearing whatsoever on the cell mapping logic.

If you specify less field values than there are cells in the space then the non-specified cells are set to blank/zero/null values as appropriate.

If you specify more field values than there are cells in the space then the additional field values are ignored.

If a key value longer than 256 bytes is specified, a fatal error will occur.

9.246 UPPERCASE

⇒ **Note:** Built-In Function Rules.

Converts a string so that all alphabetic characters are in uppercase.

For use with

LANSA for i	YES
Visual LANSA for Windows	YES
Visual LANSA for Linux	YES

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	String to change to uppercase	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Converted string	1	256		

9.247 ZIP_ADD

⇒ **Note:** Built-In Function Rules.

Allows files to be added to a .zip file. If the .zip file does not exist, it is created.

Also See

[ZIP Built-In Function Note](#)

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Zip file/path name If no path is specified, the file is created in the temporary directory. If file name does not have an extension, .zip will be appended.	1	256		
2	A	Req	Directory to use as base for zipping.	1	256		
3	L	Opt	Working list containing file names or specifications to include. This argument also applies to files in subdirectories, but not to the subdirectory names. The default is to include all. This argument is equivalent to zip -I	256	256		
4	A	Opt	Include subdirectories.	1	1		

			The default is Y. This argument is equivalent to zip -r				
5	A	Opt	Delete zipped files if zip successful. The default is N. This argument is equivalent to zip -m Note: it is not an error if files cannot be deleted.	1	1		
6	A	Opt	Compression level. 0 - 9. 0 = no compression (fastest) 9 = smallest (slowest) The default is 9. This argument is the equivalent of zip -#, where # is the number. Note: zip's default level is 6.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code OK = Zip file created successfully ER = An error was encountered NR = No matching files found	2	2		

Technical Notes

Note: If files to be added to the .zip file are already in the zip file, they will be

replaced.

9.248 ZIP_DELETE

⇒ **Note:** Built-In Function Rules.

Allows files to be deleted from a .zip file (zip -d).

Also See

[ZIP Built-In Function Note](#)

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Zip file/path name If no path is specified, the file is assumed to be in the temporary directory. If file name does not have an extension, .zip will be appended.	1	256		
2	L	Req	Working list containing file names or specifications to delete. Note: The list is case sensitive. If the filenames or file specifications do not exactly match, the files will not be deleted.	256	256		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code OK = Zip file updated successfully ER = An error was encountered NR = No matching files found	2	2		

9.249 ZIP_EXTRACT

⇒ **Note:** Built-In Function Rules.

Allows files to be extracted from a .zip file.

Also See

[ZIP Built-In Function Note](#)

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Zip file/path name If no path is specified, the file is assumed to be in the temporary directory. If file name does not have an extension, .zip will be appended.	1	256		
2	A	Req	Directory to extract to. Note: This directory will be created if necessary (and possible)	1	256		
3	L	Opt	Working list containing file names or specifications to extract. This argument also applies to files in subdirectories, but not to the subdirectory names. The default is to extract all. Note: If the list is empty, the default will be	256	256		

			used.				
4	A	Opt	<p>Overwrite existing files.</p> <p>N = Never (unzip -n)</p> <p>U = Only if newer (unzip -uo)</p> <p>Y = Always (unzip -o)</p> <p>Default is Y.</p> <p>Note: When using option U, be careful of unzipping in one time zone a zipfile created in another -- ZIP archives other than those created the BIF ZIP_ADD (or Zip 2.1 or later) contain no time zone information, and a 'newer' file from an eastern time zone may, in fact, be older.</p>	1	1		
5	L	Opt	<p>Working list containing file names or specifications to exclude. This argument also applies to files in subdirectories, but not to the subdirectory names</p> <p>The default is to exclude none.</p> <p>Note: If the list is empty, the default will be used.</p> <p>This argument is equivalent to unzip -x</p>	256	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return Code</p> <p>OK = Zip file extracted successfully</p> <p>ER = An error was encountered</p> <p>NR = No matching files</p>	2	2		

		found					
--	--	-------	--	--	--	--	--

Technical Notes

- When specifying file names or specifications to include or exclude, the forward slash (/) must always be used when specifying a path. For example, x_lansa/x_ppp/mytable.dll

9.250 ZIP_GET_INFO

⇒ **Note:** Built-In Function Rules.

Allows information about a zip file to be retrieved.

Also See

[ZIP Built-In Function Note](#)

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Zip file/path name If no path is specified, the file is assumed to be in the temporary directory. If file name does not have an extension, .zip will be appended.	1	256		
2	L	Opt	Working list containing file names or specifications to include. This argument also applies to files in subdirectories, but not to the subdirectory names. The default is to include all. The list cannot contain more than 32,767 entries. Note: If the list is empty, the default will be used.	256	256		

3	L	Opt	<p>Working list containing file names or specifications to exclude. This argument also applies to files in subdirectories, but not to the subdirectory names</p> <p>The default is to exclude none.</p> <p>The list cannot contain more than 32,767 entries.</p> <p>Note:</p> <p>If the list is empty, the default will be used.</p> <p>This argument is equivalent to zipinfo -x</p>	256	256		
4	A	Opt	<p>Positioning filename</p> <p>Note: This argument would typically only be used to continue loading the list after a previous return code of OV</p>	1	256		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	<p>Return Code</p> <p>OK = Zip file able to be read okay</p> <p>NR = No matching files found</p> <p>ER = An error was encountered</p> <p>OV = List returned completely filled, but more files in zip file than can fit in the list</p>	2	2		
2	N	Opt	<p>Total size of matching uncompressed files, in bytes</p> <p>Note: This value is only affected by the list of files to include and exclude. If this BIF is called multiple times for overflow processing, this value will not change.</p>	1	10	0	0
3	N	Opt	Number of matching files	1	10	0	0

			Note: This value is only affected by the list of files to include and exclude. If this BIF is called multiple times for overflow processing, this value will not change.				
4	L	Opt	Working list to contain file names; will include paths if they exist. List must not contain more than 32,767 entries.	256	256		
5	A	Opt	The last file name in the returned list	1	256		
6	L	Opt	Working list to contain information about the files returned in the file name list. Entry 1 in the file name list relates directly to Entry 1 in this list. This list must have the same number of entries as the file name list. Formatted: Start End Description 1 - 10 Uncompressed size (bytes) 11 - 18 Date YYYYMMDD 19 - 24 Time HHMMSS	18	24		

Technical Notes

- When specifying file names or specifications to include or exclude, the forward slash (/) must always be used when specifying a path. For example, x_lansa/x_ppp/mytable.dll
- This BIF can be used to test (zip -t) the validity of a .zip file, without returning any information, if only the first return value is specified.

9.251 ZIP_MAKE_EXE

⇒ **Note:** Built-In Function Rules.

Allows a .zip file to be converted to a self-extracting archive. When the .exe file is run, the zipped files will be extracted to the current directory.

Also See

[ZIP Built-In Function Note](#)

For use with

LANSA for i	NO
Visual LANSA for Windows	YES
Visual LANSA for Linux	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Zip file/path name If no path is specified, the file is assumed to be in the temporary directory. If file name does not have an extension, .zip will be appended.	1	256		
2	A	Opt	Exe file/path name If no path is specified, the file will be created in the temporary directory. If file name does not have an extension, .exe will be appended The default is to use the same file name as argument 1 with .exe instead of .zip	1	256		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return Code OK = Exe file created okay ER = An error was encountered	2	2		

Technical Notes

The .exe file that is created also supports some useful command line options:

- t Test the validity of the zip file contained within the self-extracting archive.
- f Extract only those files that already exist on disk and that are newer than the disk copies
- u Update existing files and create new ones if needed
- n Never overwrite existing files. By default, if a file already exists, the user is prompted
- o Always overwrite existing files
- q Quiet mode (no listing of files being extracted).
- Even quieter mode
- qq

10. Intrinsic Functions

Intrinsic Functions provide a series of facilities that can be used to manipulate individual data values. The following sections describes the use of intrinsic functions:

[10.1 Understanding Intrinsic Functions](#)

[10.2 Chaining Multiple Intrinsic Functions](#)

[10.3 Isxxxxxx Intrinsic Functions](#)

[10.4 Asxxxxxx Intrinsic Functions](#)

[10.5 Field Intrinsic Functions](#)

[10.6 Alphanumeric/String Intrinsic Functions](#)

[10.7 Boolean Intrinsic Functions](#)

[10.8 Date Intrinsic Functions](#)

[10.9 DateTime Intrinsic Functions](#)

[10.10 Time Intrinsic Functions](#)

[10.11 Large Object Intrinsic Functions](#)

[10.12 Binary Intrinsic Functions](#)

[10.13 Decimal Intrinsic Functions](#)

[10.14 Fixed Point Intrinsic Functions](#)

[10.15 Floating Point Intrinsic Functions](#)

[10.16 Integer Intrinsic Functions](#)

10.1 Understanding Intrinsic Functions

Intrinsic Functions exist for each generic class of data available in LANSAs. Thus, there are Intrinsic Functions for strings, numbers, dates, times, date times and so on. All Intrinsic Functions return a single result, and there is no need to specify the result parameter on the command line. As such, it is assumed that the result of an Intrinsic Function is the value to be used in the command.

Consider the following example

```
#com_owner.caption := 'Employee salary is ' + #salary.asstring
```

In the above example the + operator is being used to concatenate a string and a numeric field. Without Intrinsic Functions, you would need to move the value from #salary in to an alphanumeric field, and then use this to build the caption. The asstring intrinsic takes the value from #salary, converts it to a string and returns it in a result, such that it can be used in the same way as any other string in LANSAs.

Intrinsic functions are not just available when referring to specific fields. By definition, they belong to the class of data. Consider the following example code

```
Mthroutine Name(Set_Caption)
```

```
#com_owner.caption := 'Employee salary is ' + #Com_owner.Get_Salary(#Emp
```

```
Endroutine
```

```
Mthroutine Name(Get_Salary)
```

```
Define_Map For(*Input) Class(#Empno) Name(#iEmployee)
```

```
Define_Map For(*Result) Class(#Salary) Name(#oSalary)
```

```
Fetch Fields(#Salary) from_file(Pslmst) with_key(#iEmployee)
```

```
#oSalary := #Salary
```

```
Endroutine
```

In this example, the Set_Caption method does much the same as the initial

example. The major difference is that rather than the salary field being used in to construct a caption, the result of the `Get_Salary` method is used. As the `Define_Map #oSalary` is of class `#salary`, by definition it is a numeric value. This means that all numeric Intrinsic Functions can be used to manipulate the value.

Similarly, if we were to refer to a numeric property of a component, e.g. the width of a form, we could the use numeric Intrinsic Functions.

When an intrinsic function is called on an SQL NULL value, it will produce SQL NULL, unless the intrinsic function specifically deals with SQL NULL. Refer to [10.12.6 IsSqlNull](#) as an example.

[↑ 10. Intrinsic Functions](#)

10.2 Chaining Multiple Intrinsic Functions

As Intrinsic Functions by definition return a result factor, it is possible to chain multiple Intrinsic Functions on one line of code.

Consider the following requirement. Set the caption of a form to tomorrow's week day e.g Tuesday or Wednesday.

While the code itself is not entirely difficult from a logical perspective, it could require a number of different variables to store the various pieces of data as the required result is calculated. We would have to take today's date, advance it by one, taking in to consideration month and year ends etc., and then convert the result to the day of the week. Whilst this is possible using BIFs, it would require a number of lines of code and some temporary storage.

However, by using multiple Intrinsic Functions, each of which will operate on the result of the preceding function, we can keep the code concise and readable. For example

```
Mthroutine Name(Get_Tomorrow_DOW)
  Define_Map For(*Result) Class(#prim_alph) Name(#oTomorrow_DOW)

  Define Field(#Today) Type(*Date)

  #oTomorrow_DOW := #Today.Now.Adjust( 1 ).AsDayOfWeek

Endroutine
```

In the code above, the Now Intrinsic is used to set #Today to today's date. The Adjust intrinsic is then used to advance the date by one day. Finally, the result of the adjusting the date is converted to a day of the week by way of the AsDayOfWeek function.

Some intrinsic functions appear to have minimal value. For example, Pred (predecessor) subtracts 1 from a number and returns it as the result. The major use for such Intrinsic Functions is as a component of a more complex evaluation. Rather than using temporary variables, or embedding expressions, which require a further set of parentheses, we can use the intrinsic.

In the following simple example, number1 is compared to number2 or number 3, but requires embedded expressions, and consequently extra sets of parentheses.

```
If ((#Number1 - 1) = #Number2) or ((#Number1 - 1) = #Number3))
```

Endif

This can also be written using the Pred intrinsic as follows

```
If ((#Number1.pred = #Number2) or (#Number1.pred = #Number3))  
Endif
```

It is difficult to clearly demonstrate the benefit of embedded Intrinsic Functions using short examples in isolation. However, when writing complex code, any simplification that results from minimizing expression and parenthesis use, must be of benefit.

[↑ 10. Intrinsic Functions](#)

10.3 Isxxxxxx Intrinsic Functions

Many Intrinsic Function names begin with IS. This type of function will return a Boolean result, and can be thought of as a short form of an If/Else construct.

For example, the isnull intrinsic tests the value of the supplied value and returns a true if the value is blanks or zero.

Previously, testing for this situation would look like the following

```
If_Null Field(#Value)
Set Com(#Button) Enabled(True)
Else
Set Com(#Button) Enabled(False)
Endif
```

Using the intrinsic, we could write

```
If (#Value.IsNull)
Set Com(#Button) Enabled(True)
Else
Set Com(#Button) Enabled(False)
Endif
```

Clearly, this does not give any great benefit, unless combining with multiple conditions. However, we could write

```
#Button.Enabled := #Value.IsNull
```

In this example, as Isnnull returns a Boolean state, it can be applied directly to any Boolean property.

This concept can be extended to use multiple conditions, And, Or and Not operators.

[↑ 10. Intrinsic Functions](#)

10.4 Asxxxxxx Intrinsic Functions

Intrinsic functions beginning with As are convertors. That is, they convert a value from one form to another.

For example, a common requirement is to use numeric values as part of display strings for things such as messages or captions. Rather than having to use Built-In Functions to convert variables to different types, it is now possible to perform the conversion within the individual command.

In this example, the numeric value #salary is converted to a string:

```
#com_owner.caption := 'Employee salary is ' + #salary.AsString
```

[↑ 10. Intrinsic Functions](#)

10.5 Field Intrinsic Functions

You can use the field intrinsic functions to examine the characteristics of a field:

[10.5.1 FieldDecimals](#) [10.5.3 FieldType](#)

[10.5.2 FieldLength](#) [10.5.4 IsDefault](#)

[10.5.5 FieldAttributeAsString](#)

[↑ 10. Intrinsic Functions](#)

10.5.1 FieldDecimals

FieldDecimals returns the number of decimal places as specified in the field definition.

FieldDecimals only evaluates the decimal places for packed and signed fields. All other field types will return 0.

Input Parameters

None

Example

```
#Decimals := #Std_price.FieldDecimals
```

In this example, decimals would take a value of 2. Std_Price is a standard LANSAs supplied field that is defined as a packed 9, 2 in the data dictionary.

[↑ 10.5 Field Intrinsic Functions](#)

10.5.2 FieldLength

FieldLength returns the length of a field as specified in the field definition. FieldLength will return 0 for Blob and Clob field types.

Input Parameters

None

Example

```
#Length := #Std_price.FieldLength
```

In this example, #Length would take a value of 9. Std_Price is a standard LANSA supplied field that is defined as a packed 9, 2 in the data dictionary.

[↑ 10.5 Field Intrinsic Functions](#)

10.5.3 FieldType

FieldType returns the type of a field as specified in the field definition.

Input Parameters

None

Example

```
#FieldType := #Std_price.FieldType
```

In this example, #Fieldtype would take a value of Packed. Std_Price is a standard LANSAs supplied field that is defined as a packed 9, 2 in the data dictionary.

[↑ 10.5 Field Intrinsic Functions](#)

10.5.4 IsDefault

IsDefault compares the subject variable with its defined default value and returns true if the two are equal.

Input Parameters

None

Example

```
#Button.enabled := #Std_price.IsDefault
```

[↑ 10.5 Field Intrinsic Functions](#)

10.5.5 FieldAttributeAsString

Fields (PRIM_FLD) provide a method to read various properties such as Description and Label.

The available properties include:

- Description
- Label
- Heading1
- Heading2
- Heading3
- HeadingSingleLine
- HeadingWithNewLines
- HeadingWithNewLinesNoBlanks
- EditCode
- EditWord

To read a property, supply the relevant property name as a parameter to the FieldAttributeAsString() method.

Example

```
#strValue := #myField. FieldAttributeAsString( Description )
```

[↑ 10.5 Field Intrinsic Functions](#)

10.6 Alphanumeric/String Intrinsic Functions

10.6.1 AsBoolean	10.6.22 CurSize	10.6.43 OccurencesOf
10.6.2 AsDate	10.6.23 DeleteSubstring	10.6.44 PositionIn
10.6.3 AsDateTime	10.6.24 FieldDefault	10.6.45 PositionOf
10.6.4 AsDBCSTFixedChar	10.6.25 InsertString	10.6.46 Remove
10.6.5 AsDBCSSString	10.6.26 IsBoolean	10.6.47 RemoveAll
10.6.6 AsFixedChar	10.6.27 IsDate	10.6.48 RemoveCharacters
10.6.7 AsFloat	10.6.28 IsDateTime	10.6.49 Repeat
10.6.8 AsInteger	10.6.29 IsDbcs	10.6.50 Replace
10.6.9 AsNumber	10.6.30 IsFloat	10.6.51 ReplaceAll
10.6.10 AsSBCSSString	10.6.31 IsMixed	10.6.52 ReplaceSubstring
10.6.11 AsTime	10.6.32 IsNull	10.6.53 Reverse
10.6.12 AsValue	10.6.33 IsNumber	10.6.54 RightAdjust
10.6.13 BlankConcat	10.6.34 IsSbcs	10.6.55 RightMost
10.6.14 ByteTypeAt	10.6.35 IsSqlNull	10.6.56 RightTrim
10.6.15 Center and Centre	10.6.36 IsTime	10.6.57 Substitute
10.6.16 CharTypeAt	10.6.37 LastPositionIn	10.6.58 Substring
10.6.17 Concat	10.6.38 LastPositionOf	10.6.59 TranslateCharacters
10.6.18 Contains	10.6.39 LeftMost	10.6.60 Trim
10.6.19 ContainsOnly	10.6.40 LeftTrim	10.6.61 TrimBlankConcat
10.6.20 CurChars	10.6.41 LowerCase	10.6.62 TrimConcat
10.6.21 CurEbcDicSize	10.6.42 OccurencesIn	10.6.63 TrimSubstitute
		10.6.64 UpperCase

↑ 10. Intrinsic Functions

10.6.1 AsBoolean

AsBoolean converts a string to a Boolean. By default, AsBoolean expects to receive a true or false string. Any other value will result in a run-time error. Optionally, you can override the defaults and specify the true and false values to be evaluated using the TrueCaption and FalseCaption parameters.

Input Parameters

Falsecaption - Value to be converted to a Boolean state of False

Truecaption - Value to be converted to a Boolean state of True

Example

In this example, AsBoolean expects a string value of true or false:

```
#Button.enabled := #String.AsBoolean
```

This is equivalent to the following:

```
Case of_Field(#String)
```

```
When (= True)
```

```
#Button.enabled := True
```

```
When (= False)
```

```
#Button.enabled := False
```

```
Otherwise
```

```
Abort Msgtxt('String value cannot be converted to a Boolean')
```

```
Endcase
```

In this example, Asboolean expects a string value of Y or N:

```
#Button.enabled := #String.AsBoolean(N Y)
```

This is equivalent to the following:

```
Case of_Field(#String)
```

```
When (= Y)
```

```
#Button.enabled := True
```

```
When (= N)  
#Button.enabled := False
```

```
Otherwise  
Abort Msgtxt('String value cannot be converted to a Boolean')
```

```
Endcase
```

Also see

[10.6.26 IsBoolean](#)

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.2 AsDate

AsDate will return a date based on the value of the string and the specified format.

If the supplied value does not conform to the required format, the application will end with a run-time error. Use the IsDate intrinsic to test the value before attempting to convert to a date.

Input Parameters

Format - Date format expected in the numeric variable

Allowable formats are

CCYY/DD/MM

CCYY/MM/DD

CCYYDDMM

CCYYMM

CCYYMMDD

DD/MM/CCYY

DD/MM/YY

DDMMCCYY

DDMMYY

ISO

MM/DD/CCYY

MM/DD/YY

MMCCYY

MMDDCCYY

MMDDYY

MMYY

SysFmt6

SysFmt8

xYYMMDD

YY/MM/DD

YYMM

YYMMDD

Example

```
If (#String.isDate(ddmmyy)
```

```
    #Date := #string.AsDate(ddmmyy)
```

```
else
```

```
    * Error processing
```

```
Endif
```

Also see

[Date Format](#)

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.3 AsDateTime

AsDateTime will return a datetime based on the value of the string and the specified format.

If the supplied value does not conform to the required format, the application will end with a run-time error. Use the IsDatetime intrinsic to test the value before attempting to convert to a datetime.

Input Parameters

Format - Datetime format expected in the numeric variable. Allowable formats are:

CCYYDDMMHHMMSS

CCYYMMDDHHMMSS

HHMMSSDDMMCCYY

HHMMSSDDMMYY

ISO

Localized_SQL

SQL

TZ

Example

```
If (#string.IsDateTime(ccyymmddhhmmss)
```

```
#Datetime := #string.AsDateTime(ccyymmddhhmmss)
```

```
else
```

```
* Error processing
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.4 AsDBCSFixedChar

AsDBCSFixedChar returns the subject string as a DBCS string of the length specified in the TargetLength Parameter, ensuring that any DBCS characters remain fully formed and are not truncated.

Input Parameters

TargetLength – Length of the returned string

Example

```
#String40 := #String256.AsDBCSFixedChar(40)
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.5 AsDBCString

As DBCS returns the subject string ignoring all single byte (SBCS) characters.

Input Parameters

None

Example

```
#DBCS := #Mixed.AsDBCString
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.6 AsFixedChar

Alphanumeric fields only.

AsFixedChar returns a result of the same length as the definition of the subject of the intrinsic function rather than the length of the contents.

Thus, when referring to a field of length 10, AsFixedChar will always return a value of 10 bytes regardless of the number of characters the field value contains.

Input Parameters

None

Example

```
#FullName := #GiveName.AsFixedChar + #Surname
```

In this example, where Givename is an alpha 20 containing the value "Veronica", and Surname contains "Brown" the result would be 'Veronica Brown".

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.7 AsFloat

AsFloat allows a string to be handled as a floating point number. If the string contains characters that cannot be converted, the application will end with a runtime error.

Use IsFloat to test the string before using AsFloat.

Input Parameters

None

Example

```
If (#String.isFloat)
```

```
    #Float := #String.AsFloat
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.8 AsInteger

AsInteger returns the ASCII/ebcdic value of the first character in the string

Windows platform

The ASCII decimal value of the first character in the string is obtained as per the following table:

nul	0	soh	1	stx	2	etx	3	eot	4	enq	5	ack	6	bel	7
bs	8	ht	9	nl	10	vt	11	np	12	cr	13	so	14	si	15
dle	16	dc1	17	dc2	18	dc3	19	dc4	20	nak	21	syn	22	etb	23
can	24	em	25	sub	26	esc	27	fs	28	gs	29	rs	30	us	31
sp	32	!	33	"	34	#	35	\$	36	%	37	&	38	'	39
(40)	41	*	42	+	43	,	44	-	45	.	46	/	47
0	48	1	49	2	50	3	51	4	52	5	53	6	54	7	55
8	56	9	57	:	58	;	59	<	60	=	61	>	62	?	63
@	64	A	65	B	66	C	67	D	68	E	69	F	70	G	71
H	72	I	73	J	74	K	75	L	76	M	77	N	78	O	79
P	80	Q	81	R	82	S	83	T	84	U	85	V	86	W	87
X	88	Y	89	Z	90	[91	\	92]	93	^	94	_	95
`	96	a	97	b	98	c	99	d	100	e	101	f	102	g	103
h	104	i	105	j	106	k	107	l	108	m	109	n	110	o	111
p	112	q	113	r	114	s	115	t	116	u	117	v	118	w	119
x	120	y	121	z	122	{	123	 	124	}	125	~	126	del	127

IBM i platform

If this intrinsic function is run on IBM i, it will return a value according to the table in www.astrodigital.org/digital/ebcdic.html. (You can ignore the Hex column in the table).

Input Parameters

None

Examples

For Windows

If #String is "a", #Integer will be 97:

```
#Integer := #String.AsInteger
```

If #String is "}", #Integer will be 125

```
#Integer := #String.AsInteger
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.9 AsNumber

Asnumber will convert a string to a number. If the string contains characters that cannot be converted, the application will end with a run-time error.

Use IsNumber to test the string before using AsNumber.

Input Parameters

None

Example

```
If (#String.isNumber)
```

```
    #number := #string.AsNumber
```

```
else
```

```
    * Error processing
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.10 AsSBCSString

AsSBCSString returns the subject string ignoring all double byte (DBCS) characters.

Input Parameters

None

Example

```
#SBCS := #Mixed.AsSBCSString
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.11 AsTime

AsTime will return a time based on the value of the string and the specified format.

If the supplied value does not conform to the required format, the application will end with a run-time error. Use the IsTime intrinsic to test the value before attempting to convert to a time.

Input Parameters

Format - Time format expected in the numeric variable. Allowable formats are:

HHMMSS

HHsMMsSS

ISO

Example

```
If (#string.IsTime(hhmmss))
```

```
#Time := #string.AsTime(hhmmss)
```

```
else
```

```
* Error processing
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.12 AsValue

The AsValue() intrinsic method can be used to specify a value to be returned from a field if it contains an SQL Null.

See [Intrinsic Method .AsValue](#).

Input Parameters

The value to be returned instead of an SQLNull.

Example

Rather than having to test as below whether the value of an object is AAA or an SQL null:

```
If ((#Std_obj = AAA) *or (#Std_obj.IsSqlNull)
```

```
Endif
```

You can use AsValue:

```
If (#Std_obj.AsValue( AAA ) = AAA)
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.13 BlankConcat

BlankConcat concatenates up to 10 strings, inserting a blank between each parameter.

All trailing spaces are trimmed. Leading spaces are left as is.

Input Parameters

String1 - String to be concatenated

String2 - String to be concatenated

String3 - String to be concatenated

String4 - String to be concatenated

String5 - String to be concatenated

String6 - String to be concatenated

String7 - String to be concatenated

String8 - String to be concatenated

String9 - String to be concatenated

String10 - String to be concatenated

Example

```
#Com_owner.Caption := #Firstname.BlankConcat(#Surname)
```

If Firstname contained Veronica and Surname contained Brown, the result would be:

Veronica Brown

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.14 ByteTypeAt

ByteTypeAt tests the type of byte in a string at a given index. This allows a string to be tested for DBCS and SBCS information. Possible results are:

Sbcs - Single byte

Dbcs1 - First byte of a double byte character

Dbcs2 - Second byte of a double byte character

ShiftIn - Shiftin byte for an EBCDIC DBCS string

ShiftOut - Shiftout byte for an EBCDIC DBCS string

Input Parameters

Index - Byte position within the supplied string

Example

This Example tests the first character of the supplied string for an EBCDIC shiftout

```
If (#String.ByteTypeAt( 1 ) = ShiftOut)
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.15 Center and Centre

Center allows a string to be centered within a given length, and will optionally pad the result with a supplied character. Leading and trailing spaces are significant and will be evaluated as part of the string to center.

If the string is longer than the target variable, the first n bytes of the string are used. Where the result of centering the string results in an uneven number of bytes, the extra byte will be allocated to the right-hand side.

Typically, centering is used to center a value within a target variable. By using the Length parameter you can control how the string is centered and padded.

Center can only be used to center a string within a target string, which does not guarantee that the result will be visually centered in a Windows run-time environment.

Input Parameters

Length - Length of the string to be centered in

Pad - Character to pad either side of the centered string.

Example

In this example, if string is a 40 byte variable that contains a value of "Centered Text", the result is "***Centered Text***". The remaining 20 bytes of #string will be null:

```
#Target := #string.Center(20 '*')
```

In this example, where string is a 20 byte variable that contains a value of "Centered Text", the result would be " Centered Text ". This is a typical centering scenario where the length of the target governs the centering of the text:

```
#Target := #string.Center( #Target.FieldLength)
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.16 CharTypeAt

CharTypeAt is similar to ByteTypeAt in that it tests the type of byte in a string at a given index. This allows a string to be tested for DBCS and SBCS information. The difference is that it only ever returns SBCS and DBCS.

Possible results are:

SBCS - Single byte character

DBCS - Double byte character

Input Parameters

Index - Byte position within the supplied string

Example

If (#String.CharTypeAt(1) = DBCS)

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.17 Concat

Concat concatenates up to 10 strings. Trailing spaces on the last string are trimmed. Leading spaces are left as is.

Input Parameters

String1 - String to be concatenated

String2 - String to be concatenated

String3 - String to be concatenated

String4 - String to be concatenated

String5 - String to be concatenated

String6 - String to be concatenated

String7 - String to be concatenated

String8 - String to be concatenated

String9 - String to be concatenated

String10 - String to be concatenated

Example

In this example, if Firstname contained '**Veronica** ' and Surname contained '**Brown** ', the result would be:

Veronica Brown

```
#Com_owner.Caption := #Firstname.Concat(#Surname)
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.18 Contains

Contains returns a true if the string contains the specified search value. The function is case sensitive.

Input Parameters

String - String to be located in the subject of the intrinsic

Startposition - Character position at which to start looking for the string

Example

```
#Button.Enabled := #String.Contains( #Search )
```

To avoid case sensitivity issues, use the Uppercase or Lowercase intrinsic functions to ensure matching cases for both strings:

```
#Button.Enabled := #String.Uppercase.Contains( #Search.Uppercase )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.19 ContainsOnly

ContainsOnly compares the contents of the subject string against the characters supplied in the String parameter and returns a true if no other characters are found in the subject.

Input Parameters

String – Characters to compare against the subject

Example

In the example below, the string is tested to ensure that it only contains the digits 0 through 9 and a blank. If other characters are used in #String, the function returns False.

```
If (#String.ContainsOnly( " 0123456789" ))
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.20 CurChars

CurChars returns the number of characters in a string.

In a single byte environment, CurChars and Cursize will return the same result.

In a DBCS environment each character requires two bytes, so although a string may be 8 bytes long, the number of characters will be 4 in ASCII and 3 in ECBDIC. Two bytes will be used by the shift in and shift out bytes.

Input Parameters

None

Example

In this example, if string contained "Sample", number would be populated with 6:

```
#Number := #String.CurChars
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.21 CurEbcDicSize

CurEbcDicSize returns the number of bytes a string would require if it was running in an EBCDIC environment.

In a DBCS environment shift in and shift out bytes are used for DBCS values. These bytes are not used in ASCII. Thus, a string that may be short enough to fit in a variable in Windows may not fit when running on the IBM i.

Input Parameters

None

Example

```
#Number := #String.CurEbcDicSize
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.22 CurSize

CurSize returns the number of bytes in a string.

In a single byte environment, CurSize and CurChars will return the same result.

In a DBCS environment each character requires two bytes. On an ASCII system, a three character DBCS string will require 6 bytes. On the IBM i, which uses EBCDIC, the shift in and shift out bytes add 2 to all byte length calculations.

Input Parameters

None.

Example

```
#Number := #String.CurSize
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.23 DeleteSubstring

DeleteSubstring deletes the characters in a string from the specified start position as far as the specified length. If a length is not specified, all characters after the start position will be deleted.

Input Parameters

StartPosition - Character at which the substring to be deleted starts.

Length - The number of characters to be deleted.

Example

In this example, if #String contained "abcd", the caption would be "acd":

```
#Com_owner.Caption := #String.DeleteSubstring( 2 1)
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.24 FieldDefault

FieldDefault can be used to set a field to its default value.

Input Parameters

None

Example

```
define field(#string) type(*string) length(12) default("qwerty")
#string := "hello"
```

* the next line will set #string to "qwerty"

```
#string := #string.FieldDefault
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.25 InsertString

InsertString allows you to embed other strings within a target variable at a specified position identified by the At parameter.

Input Parameters

String - String to be inserted

At - Position to insert the string

Pad - Character used to pad the result if the At parameter is beyond the end of the string

Example

In this example, if string contained "abcdefg", the result would be "abcABCDefg":

```
#Com_owner.Caption := #String.InsertString( "ABCD" 4)
```

In this example, if string contained "abcdefg", the At parameter is beyond the end of the string, the result would be "abcdefg**ABCD".:

```
#Com_owner.Caption := #String.InsertString( "ABCD" 10 "**")
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.26 IsBoolean

IsBoolean tests a string to determine whether it can be used as a boolean. The Truecaption and Falsecaption parameters define the allowable values for the Boolean.

IsBoolean will often be used before [10.6.1 AsBoolean](#) to so that you can trap a potential run-time error.

Input Parameters

Falsecaption - Value to represent a Boolean state of False

Truecaption - Value to represent a Boolean state of True

If no input parameters are specified, the function expects to receive the strings True or False.

Example

In this example, isBoolean expects a string value of N or Y:

```
If (#String.IsBoolean(N Y))  
  
    #Button.Enabled := #String.IsBoolean(N Y)  
  
Else  
  
    * Error processing  
  
Endif
```

This is equivalent to writing the following:

```
Case of _Field(#String)  
  
    When (= Y)  
        #Button.enabled := True  
  
    When (= N)  
        #Button.enabled := False  
  
    Otherwise  
        Abort Msgtxt('String value cannot be converted to a Boolean')
```

Endcase

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.27 IsDate

IsDate will return true if the string can be converted to a valid date in the specified format. IsDate will often be used before AsDate to better handle potential date errors.

Input Parameters

Format - Date format expected in the string variable. Allowable formats are:

CCYY/DD/MM

CCYY/MM/DD

CCYYDDMM

CCYYMM

CCYYMMDD

DD/MM/CCYY

DD/MM/YY

DDMMCCYY

DDMMYY

ISO

MM/DD/CCYY

MM/DD/YY

MMCCYY

MMDDCCYY

MMDDYY

MMYY

SysFmt6

SysFmt8

xYYMMDD

YY/MM/DD

YYMM

YYMMDD

Example

```
If (#String.IsDate(DDMMYY))
```

```
#Date := #String.Asdate(ddmmyy)
```

else

* Error processing

Endif

Also see

[Date Format](#)

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.28 IsDateTime

IsDateTime will return true if the string can be converted to a valid datetime in the specified format. IsDateTime will often be used before AsDateTime to better handle potential errors.

Input Parameters

Format - Datetime format expected in the string variable. Allowable formats are:

CCYYDDMMHHMMSS

CCYYMMDDHHMMSS

HHMMSSDDMMCCYY

HHMMSSDDMMYY

ISO

Localized_SQL

SQL

TZ

Example

```
If (#string.IsDateTime(ccyymmddhhmmss)
```

```
#Datetime := #string.Asdatetime(ccyymmddhhmmss)
```

```
else
```

```
* Error processing
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.29 IsDbcs

IsDbcs returns true if all of the characters in a string are double byte.

Input Parameters

None

Example

```
#Buttons.Enabled := #string.IsDbcs
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.30 IsFloat

IsFloat examines whether a string's value can be handled as a floating point number. It returns a boolean value - true if the value is a floating point number, false if it is not. It is a good idea to use IsFloat as a test before calling AsFloat.

Input Parameters

None.

Example

```
If cond(#String.isFloat)
  #Float := #String.AsFloat
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.31 IsMixed

IsMixed returns true if there is a mixture of single byte and double byte characters in the supplied string.

Input Parameters

None

Example

```
#Buttons.Enabled := #string.IsMixed
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.32 IsNull

IsNull tests a string variable and returns a true if it contains a *null value.

Input Parameters

None

Example

```
#Button.enabled := #String.IsNull
```

The *null value for a numeric variable is *blanks.

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.33 IsNumber

IsNumber returns true if the supplied string can be converted to a number.

IsNumber will often be used before AsNumber to better handle potential errors.

Input Parameters

None

Example

```
If (#String.IsNumber)
```

```
    #number := #String.AsNumber
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.34 IsSbcs

IsSbcs returns true if all of the characters in a string are single byte characters.

Input Parameters

None

Example

```
#Buttons.Enabled := #string.IsSbcs
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.35 IsSqlNull

IsSqlNull tests a string variable and returns true if it contains an SQL Null.

Input Parameters

None

Example

```
#Button.enabled := #String.IsSqlNull
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.36 IsTime

IsTime will return true if the string can be converted to a valid time in the specified format.

IsTime will often be used before AsTime to better handle potential errors.

Input Parameters

Format - Time format expected in the string variable. Allowable formats are:

HHMMSS

HHsMMsSS

ISO

Example

```
If (#string.IsTime(hhmmss))
```

```
    #Time := #string.AsTime(hhmmss)
```

```
else
```

```
    * Error processing
```

```
Endif
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.37 LastPositionIn

LastPositionIn returns the last position of the subject string in a string. If the string is not found, the result will be 0.

LastPositionIn is case sensitive.

Input Parameters

String - String to be searched in.

Example

In this example, if #String contained M, the result would be 13:

```
#LastPosition := #String.LastPositionIn( 'ABCDEFGHJKLMNOPQRSTUVWXYZ')
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.38 LastPositionOf

LastPositionOf returns the last position of a string in the subject string. If the string is not found, the result will be 0.

LastPositionOf is case sensitive.

Input Parameters

String - String to be searched for.

Example

In this example, if #String contained ABCDEFGHIJKLMNOPQRSTUVWXYZ, the result would be 13:

```
#LastPosition := #String.LastPositionOf( M )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.39 LeftMost

LeftMost returns the first n characters of a string. If the string does not have enough characters, the remaining space can be padded.

Input Parameters

Characters – Number of characters to be returned

Pad - Character to be used to fill remaining space

Example

In this example, if #String contained ABCDEFGHIJKLMNOPQRSTUVWXYZ, the result would be ABCDEFGHIJKLM:

```
#Com_owner.Caption := #String.LeftMost( 13 )
```

In this example, if #String contained ABCDEFGHI the result would be ABCDEFGHI****:

```
#Com_owner.Caption := #String.LeftMost( 13 '*' )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.40 LeftTrim

LeftTrim can be used to remove leading blanks, or other characters, from a string.

The OfText parameter can contain more than one character.

Input Parameters

OfText - Character or characters to be trimmed. The default is a blank

Example

In this example, if #String contained ' ABCDE', the result would be 'ABCDE':

```
#Com_owner.Caption := #String.LeftTrim
```

In this example, if #String contained AAA the result would be A. After the first AA has been removed from the string, only a single A remains that does not match the OfText parameter value:

```
#Com_owner.Caption := #String.LeftTrim( AA )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.41 LowerCase

Lowercase returns the supplied string with all characters converted to lowercase.

Input Parameters

None

Example

In this example, if #String contained 'ABCDE', the result would be 'abcde':

```
#String := #String.Lowercase
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.42 OccurencesIn

OccurencesIn returns the number of times the string can be found in the string supplied in the String parameter, starting from the character specified in the Startposition parameter.

OccurencesIn is case sensitive

Input Parameters

String - String to be searched

StartPosition - Character to start searching from

Example

In this example, if #String contained 'ABC', the result would be 3:

```
#Occurences := #String.OccurencesIn( 'ABCDEABCDEABCDE' )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.43 OccurencesOf

OccurencesOf returns the number of times the string supplied in the String parameter can be found in a string, starting from the character specified in the Startposition parameter.

OccurencesOf is case sensitive

Input Parameters

String - String to be searched for

StartPosition - Character to start searching from

Example

In this example, if #String contained 'ABCDEABCDEABCDE', the result would be 3:

```
#Occurences := #String.OccurencesOf( 'ABC' )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.44 PositionIn

PositionIn returns character position of the first occurrence of the String in the supplied string parameter after the StartPosition.

PositionIn is case sensitive

Input Parameters

String - String to be searched

StartPosition - Character to start searching from

Example

In this example, if #String contained 'EAB', the result would be 5:

```
#Position := #String.PositionIn( 'ABCDEABCDEABCDE' )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.45 PositionOf

PositionOf returns the character position of the first occurrence in the supplied String parameter of the String, after the StartPosition.

PositionOf is case sensitive

Input Parameters

String - String to be searched

StartPosition - Character to start searching from

Example

In this example, if #String contained 'ABCDEABCDEABCDE', the result would be 5:

```
#Position := #String.PositionOf( 'EAB' )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.46 Remove

Remove removes the first occurrence of the string supplied in the Object parameter

Input Parameters

Object – String to be removed from the subject

Example

In this example, if #String contained the value "CCBBAA", the result would be "CCBBA"

```
#String := #String.Remove(A)
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.47 RemoveAll

RemoveAll removes all occurrences of the string supplied in the Object parameter. This is similar to the RemoveCharacters intrinsic, except that RemoveAll searches the subject string for instances of the entire search string rather than individual characters. RemoveAll is case sensitive.

Input Parameters

Object – String to be removed from the subject

Example

In this example, if #String contained the value "HABERDASHERY", the result would be "HABERERY"

```
#String := #String.RemoveAll( "DASH" )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.48 RemoveCharacters

RemoveCharacters removes all occurrences of the characters supplied in the Object parameter. This is similar to the RemoveAll intrinsic, except that RemoveCharacters searches the subject string for individual characters rather than instances of the entire search string. RemoveCharacters is case-sensitive.

Input Parameters

Object – String to be removed from the subject

Example

In this example, if #String contained the value "ADACABA", the result would be "DCB"

```
#String := #String.RemoveCharacters("A")
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.49 Repeat

Repeat returns the string repeated the specified number of times.

Input Parameters

Times - Number of times to repeat the string. This number must be greater than zero.

Example

In this example, if #String contained 'ABCDE' the result would be 'ABCDEABCDE':

```
#Com_owner.Caption := #String.Repeat( 2 )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.50 Replace

Replace replaces the first instance of the value specified in the Object parameter with the value specified in the Replacement parameter

Input Parameters

Object – Value to be replaced

Replacement – Value to be used in place of the Object parameter value

Example

```
#Com_owner.Caption := #String.Replace( " " "_" )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.51 ReplaceAll

ReplaceAll replaces all instances of the value specified in the Object parameter with the value specified in the Replacement parameter.

Input Parameters

Object – Value to be replaced

Replacement – Value to be used in place of the Object parameter value

Example

Replace all spaces in #string with an underscore

```
#Com_owner.Caption := #String.ReplaceAll( " " "_" )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.52 ReplaceSubstring

ReplaceSubstring replaces the characters from the StartPosition for the specified length with the contents of the With parameter.

If the length exceeds the available characters, the remainder is ignored.

Input Parameters

StartPosition - Character at which to start replacing

Length - Number of characters to replace

With - Replacement string

Pad - Pad character to be used when the StartPosition is beyond the length of the string.

Example

In this example, if #String contained 'ABCDE', the result would be 'XYZDE':

```
#Com_owner.Caption := #String.ReplaceSubstring( 1 3 'XYZ' )
```

In this example, if #String contained 'ABCDE', the result would be 'AXYZCDE':

```
#Com_owner.Caption := #String.ReplaceSubstring( 2 1 'XYZ' )
```

In this example, if #String contained 'ABCDE', the result would be 'ABCDE****XYZ':

```
#Com_owner.Caption := #String.ReplaceSubstring( 10 1 'XYZ' '*' )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.53 Reverse

Reverse returns the string reversed end to end.

Input Parameters

None

Example

In this example, if #String contained 'ABCDE' the result would be 'EDCBA':

```
#Com_owner.Caption := #String.Reverse
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.54 RightAdjust

RightAdjust shifts the value in a string as far to the right as allowed by either the length parameter or the string's length, whichever is smaller. The left of the value is filled with the contents of the pad parameter, or blank spaces if the pad isn't provided.

Input Parameters (both optional)

Length - the length within which to right-adjust the string's value.

Pad - a character used to pad space created to the left of the value.

Examples

```
define field(#source) type(*string) length(10)
define field(#target) type(*string) length(10)
```

* in this example, #target will be set to " abc"

```
#source := "abc"
```

```
#target := #source.RightAdjust
```

* in this example, #target will be set to " abc"

```
#target := #source.RightAdjust( 5 )
```

* in this example, #target will be set to "xxabc"

```
#target := #source.RightAdjust( 5, "x" )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.55 RightMost

RightMost returns the 'n' rightmost characters from the string.

Input Parameters

Characters - Number of characters to retrieve.

Pad - Pad character to be used if the number of characters exceeds the length of the string.

Example

In this example, if #String contained 'ABCDE' the result would be 'CDE':

```
#Com_owner.Caption := #String.RightMost(3)
```

In this example, if #String contained 'ABCDE' the result would be

'*****ABCDE':

```
#Com_owner.Caption := #String.RightMost(10 '*')
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.56 RightTrim

RightTrim can be used to remove trailing blanks, or other characters, from a string.

The OfText parameter can contain more than one character.

Input Parameters

OfText - Character or characters to be trimmed. The default is a blank

Example

In this example, if #String contained 'ABCDE ', the result would be 'ABCDE':

```
#Com_owner.Caption := #String.RightTrim
```

In this example, if #String contained AAA the result would be A. After the first AA has been removed from the string, only a single A remains that does not match the OfText parameter value:

```
#Com_owner.Caption := #String.RightTrim( AA )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.57 Substitute

Substitute allows you to replace text at a series of predetermined points in a string, identified by &1, &2...&9.

Input Parameters

String1 - Substitution value

String2 - Substitution value

String3 - Substitution value

String4 - Substitution value

String5 - Substitution value

String6 - Substitution value

String7 - Substitution value

String8 - Substitution value

String9 - Substitution value

Example

In this example, *MtxtCust01 is a multilingual variable containing the following:

```
"&1 &2 has a limit of $&3"
```

#Givename, #Surname and #Limit have values of Veronica, Brown and 2000 respectively.

The result at run-time would be:

Veronica Brown has a limit of \$2000.

```
#Com_owner.caption := *MtxtCust01.Substitute(#Givename #Surname #Limit
```

Note: If two ampersands appear together in a string, they are reduced to a single ampersand and not considered for substitution.

For example:

```
#str1 := "&1&&2"
```

```
#str2 := #str1.Substitute( "a" "b" )
```

#str2 will equal "a&2"

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.58 Substring

Substring returns a section of the string starting at the specified StartPosition for a length of the specified Length. If this combination exceeds the available string length, Pad can be used to provide a pad character.

Input Parameters

StartPosition - Character at which to start the Substring

Length - number of characters to substring

Pad - Pad character to be used when length exceeds available string

Example

In this example, if #String contained 'ABCDE', the result would be 'ABCD':

```
#Com_owner.Caption := #String.Substring( 1 4)
```

In this example, if #String contained 'ABCDE', the result would be 'ABCDE***':

```
#Com_owner.Caption := #String.Substring( 1 8 '*')
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.59 TranslateCharacters

TranslateCharacters is used to substitute characters in the subject string with characters in the 'To' string, using characters in the 'From' string as a key. Whenever a character in the subject string matches a character in the 'From' string it will be replaced with the equivalent character in the 'To' string.

Input Parameters

From - a string containing the characters to search for in the subject string

To - a string containing the characters to replace in the subject string

Examples

The following assume that #String contains 'QWERTY':

* #String2 will be set to 'YTREWQ'

```
#String2 := #String.TranslateCharacters( 'QWERTY', 'YTREWQ' )
```

* #String2 will be set to 'qWeRtY'

```
#String2 := #String.TranslateCharacters( 'QET', 'qet' )
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.60 Trim

Trim can be used to remove leading and trailing blanks, or other characters, from a string.

The OfText parameter can contain more than one character.

Input Parameters

OfText - Character or characters to be trimmed. The default is a blank

Example

In this example, if #String contained ' ABCDE ', the result would be 'ABCDE':

```
#Com_owner.Caption := #String.Trim
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.61 TrimBlankConcat

TrimBlankConcat concatenates up to 10 strings, removing trailing blanks, and inserting a blank between each parameter.

Input Parameters

String1 - String to be concatenated

String2 - String to be concatenated

String3 - String to be concatenated

String4 - String to be concatenated

String5 - String to be concatenated

String6 - String to be concatenated

String7 - String to be concatenated

String8 - String to be concatenated

String9 - String to be concatenated

String10 - String to be concatenated

Example

If #Firstname contained ' Veronica ' and #Surname contained ' Brown ', the result would be 'Brown Veronica':

```
#Com_owner.Caption := #Surname.TrimBlankConcat(#Firstname)
```

This is equivalent to writing:

```
#Com_owner.Caption := #Surname.Trim + ' ' + #Firstname.Trim
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.62 TrimConcat

TrimConcat concatenates up to 10 strings, removing all trailing blanks.

Input Parameters

String1 - String to be concatenated

String2 - String to be concatenated

String3 - String to be concatenated

String4 - String to be concatenated

String5 - String to be concatenated

String6 - String to be concatenated

String7 - String to be concatenated

String8 - String to be concatenated

String9 - String to be concatenated

String10 - String to be concatenated

Example

If #Firstname contained ' **Veronica** ' and #Surname contained ' **Brown** ', the result would be ' **Brown Veronica**':

```
#Com_owner.Caption := #Surname.TrimConcat( #Firstname )
```

This is equivalent to writing:

```
#Com_owner.Caption := #Surname.RightTrim + #Firstname.RightTrim
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.63 TrimSubstitute

TrimSubstitute allows you to replace text at a series of predetermined points in a string, identified by &1, &2...&9, trimming all trailing blanks.

Input Parameters

String1 - Substitution value

String2 - Substitution value

String3 - Substitution value

String4 - Substitution value

String5 - Substitution value

String6 - Substitution value

String7 - Substitution value

String8 - Substitution value

String9 - Substitution value

Example

In this example, *MtxtCust01 is a multilingual variable containing the following:

```
"&1 &2 has a limit of $&3"
```

#Givename, #Surname and #Limit have values of 'Veronica ', 'Brown ' and '2000' respectively.

The result at run-time would be:

Veronica Brown has a limit of \$2000.

```
#Com_owner.caption := *MtxtCust01.Substitute(#Givename #Surname #Limit
```

Note: If two ampersands appear together in a string, they are reduced to a single ampersand and not considered for substitution.

For example:

```
#str1 := "&1&&2"
```

```
#str2 := #str1.Substitute( "a" "b" )
```

#str2 will equal "a&2"

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.6.64 UpperCase

Uppercase returns the supplied string with all characters converted to uppercase.

Input Parameters

None

Example

In this example, if #String contained 'abcde', the result would be 'ABCDE':

```
#String := #String.Uppercase
```

[↑ 10.6 Alphanumeric/String Intrinsic Functions](#)

10.7 Boolean Intrinsic Functions

In the examples for the following functions, simple names have been used to identify individual variables purely for the purposes of clarity. They are not representative of any variable in your LANSAs repository.

[10.7.1 And](#) [10.7.6 IsNull](#)

[10.7.2 AsNumber](#) [10.7.7 IsSqlNull](#)

[10.7.3 AsString](#) [10.7.8 IsTrue](#)

[10.7.4 AsValue](#) [10.7.9 Not](#)

[10.7.5 IsFalse](#) [10.7.10 Or](#)

Also see

[↑ 10. Intrinsic Functions](#)

10.7.1 And

And checks the value of two Boolean values. If both are true, the result will be True, otherwise, it will be false.

Input Parameters

With - Name of the second variable to be tested

Example

```
#Button.Enabled := #Boolean1.And(#Boolean2)
```

This is equivalent to writing

```
If (#Boolean1 *and #boolean2)
```

```
#Button.Enabled := True
```

```
Else
```

```
#Button.Enabled := False
```

```
Endif
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.2 AsNumber

AsNumber returns a Boolean variable as a number. If false, the result will be 0. If true, it will be 1

Input Parameters

None

Example

```
#Number := #Boolean.AsNumber
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.3 AsString

AsString returns a Boolean variable as a string. If false, the result will be "False". If true, it will be "True".

Input Parameters

None

Example

```
#Number := #Boolean.AsString
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.4 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable values for a Boolean variable are 0, 1, True and False

Example

Rather than having to test as below

```
If ((#Boolean = True) *or (#Boolean.IsSqlNull)
```

```
Endif
```

AsValue allows the following:

```
If (#Boolean.AsValue(True))
```

```
Endif
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.5 IsFalse

IsFalse tests a Boolean variable and returns true if the Boolean is False

Input Parameters

None

Example

```
#Button.Enabled := #Boolean.IsFalse
```

This is equivalent to writing

```
If (#Boolean = False)
```

```
#Button.Enabled := True
```

```
Else
```

```
#Button.Enabled := False
```

```
Endif
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.6 IsNull

IsNull tests a Boolean variable and returns a true if it contains a *null value.

Input Parameters

None

Example

```
#Button.enabled := #Boolean.IsNull
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.7 IsSqlNull

IsSqlNull tests a Boolean and returns true if it contains an SQL null

Input Parameters

None

Example

```
#Button.enabled := #Boolean.IsSqlNull
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.8 IsTrue

IsTrue tests a Boolean variable and returns true if the Boolean is False

Input Parameters

None

Example

```
#Button.Enabled := #Boolean.IsTrue
```

This is equivalent to writing

```
If (#Boolean = True)
```

```
#Button.Enabled := True
```

```
Else
```

```
#Button.Enabled := False
```

```
Endif
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.9 Not

Not returns the opposite value of a Boolean variable. Thus, a true will be returned as a false, and vice versa

Input Parameters

None

Example

```
#Button.Enabled := #Boolean.Not
```

This is equivalent to writing

```
If (#Boolean = True)
```

```
#Button.Enabled := False
```

```
Else
```

```
#Button.Enabled := True
```

```
Endif
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.7.10 Or

Or checks the value of two Boolean values. If either is true, the result will be True, otherwise, it will be false.

Input Parameters

With - Name of the second variable to be tested

Example

```
#Button.Enabled := #Boolean1.Or(#Boolean2)
```

This is equivalent to writing

```
If (#Boolean1 *or #boolean2)
```

```
#Button.Enabled := True
```

```
Else
```

```
#Button.Enabled := False
```

```
Endif
```

[↑ 10.7 Boolean Intrinsic Functions](#)

10.8 Date Intrinsic Functions

In the examples for the following functions, simple names have been used to identify individual variables purely for the purposes of clarity. They are not representative of any variable in your LANSAs repository.

10.8.1 Adjust 10.8.6 AsNumber 10.8.11 IsNull
10.8.2 AsDateTime 10.8.7 AsString 10.8.12 IsSqlNull
10.8.3 AsDayofWeek 10.8.8 AsValue 10.8.13 Month
10.8.4 AsDays 10.8.9 Day 10.8.14 Now
10.8.5 AsDisplayString 10.8.10 Difference 10.8.15 Year

↑ 10. Intrinsic Functions

10.8.1 Adjust

Adjust increments or decrements a date by the number of days specified

Input Parameters

Adjustment - Value to increment or decrement the date

Example

```
#Tomorrow := #Today.Adjust(1)
```

or

```
#Yesterday := #Today.Adjust(-1)
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.2 AsDateTime

AsDateTime returns a date variable as a datetime.

Input Parameters

Time - time variable to be appended to the date

Example

In this Example, a #Date and #Time are combined to produce a DateTime Variable:

```
#DateTime := #Date.AsDateTime(#Time)
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.3 AsDayofWeek

AsDayofWeek returns the equivalent day of the week for the supplied date.

Input Parameters

None

Example

In this example, if Today contained the value 01-01-2005, #dayofweek would become Saturday for English systems:

```
#DayName := #Today.AsDayofWeek
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.4 AsDays

AsDays returns the number of days since 01-01-0000 for the supplied date.

Input Parameters

None

Example

```
#NoofDays := #Today.AsDays
```

A date of 2004-12-31 will return result of 731945.

[↑ 10.8 Date Intrinsic Functions](#)

10.8.5 AsDisplayString

AsDisplayString returns the supplied date formatted using one of the available supplied date formats

Input Parameters

Format - Required format of the data. Note that the month is multilingual. SYSFMT dates are in operating system format.

Available formats, and the resulting display string are listed below. Examples use a date of 2004-12-31.

See Format Note	Display String
CCYYDDMM	20043112
CCYYMM	200412
CCYYMMDD	20041231
CCYYsDDsMM	2004/31/12
CCYYsMMsDD	2004/12/31
Fri	Fri
DDDDDDDDDD	Friday
DDMMCCYY	31122004
DDMMMCCYY	31Dec2004
DDMMYY	31Dec04
DDMMYY	311204
DDsMMsCCYY	31/12/2004
DDsMMsYY	31/12/04
DDXXbMMMMMMMMMMbCCYY	31st December 2004
DDXXbMMMMMMMMMMbYY	31st December 04
ISO	2004-12-31

1	ML_DDDDDDDDD	Friday
1	ML_DDbMMMMMMMMMbYY	31 December 04
1	ML_DDbMMMMMMMMMbCCYY	31 December 2004
	MMCCYY	122004
	MMDDCCYY	12312004
	MMDDYY	123104
1	MMMMMMMMMMM	December
	MMsDDsCCYY	12/31/2004
	MmsDDsYY	12/31/04
	MMYY	1204
	SYSFMT6	311204
	SYSFMT8	31122004
	XYYMMDD	20043112
	YYMM	0412
	YYMMDD	041231
	YysMMsDD	04/12/31

Example

```
#Com_owner.Caption := #Today.AsDisplayString(DDMMCCYY)
```

Also see

[Date Format](#)

[↑ 10.8 Date Intrinsic Functions](#)

10.8.6 AsNumber

AsNumber returns the supplied date, formatted as specified, as a number.

Input Parameters

Format - Required format of the data

Available formats, and the resulting display string are listed below. Examples use a date of 2004-12-31.

Format	Display Number
CCYYDDMM	20043112
CCYYMM	200412
CCYYMMDD	20041231
DAYS	731945
DDMMCCYY	31122004
DDMMYY	311204
MMCCYY	122004
MMDDCCYY	12312004
MMDDYY	123104
MMYY	1204
SYSFMT6	311204
SYSFMT8	31122004
XYYMMDD	20043112
YYMM	412
YYMMDD	41231

Example

```
#Com_owner.Caption := #Today.AsNumber(CCYYDDMM)
```

Also see

[Date Format](#)

[↑ 10.8 Date Intrinsic Functions](#)

10.8.7 AsString

AsString returns the date as a string.

Input Parameters

None

Example

```
#Com_owner.Caption := #Today.AsString
```

A date of 2004-31-12 will be returned as a string as follows 2004-31-12

[↑ 10.8 Date Intrinsic Functions](#)

10.8.8 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable values for a date variable are any valid dates

Example

The IF below tests for the date as SQLnull or today's date

```
If ((#Date.IsSqlNull) *or (#Date = #Date.now))
```

```
Endif
```

Using AsValue it can be coded as follows

```
If (#Date.AsValue(#Date.now) = #Date.now)
```

```
Endif
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.9 Day

Day returns the day portion of a date.

Input Parameters

None

Example

In this example, a date of 2004-12-31 would return a value of 31 to Day:

```
#Day := #Today.Day
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.10 Difference

Difference returns the number of days between the subject and object days.

Input Parameters

Object - Date to be compared against

Example

The example below will return a value of 1. That is, tomorrow is 1 day after or greater than today.

```
#DaysDiff := #Tomorrow.Difference(#today)
```

Reversing the parameters, as below, will give a result of -1. That is, today is one day before or less than tomorrow.

```
#DaysDiff := #Today.Difference(#Tomorrow)
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.11 IsNull

IsNull tests a date variable and returns a true if it contains a *null value.

Input Parameters

None

Example

```
#Button.enabled := #Date.IsNull
```

The *null value for a date is 1900-01-01.

[↑ 10.8 Date Intrinsic Functions](#)

10.8.12 IsSqlNull

IsSqlNull tests a date variable and returns true if it contains an SQL Null

Input Parameters

None

Example

```
#Button.enabled := #Date.IsSqlNull
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.13 Month

Month returns the month portion of a date.

Input Parameters

None

Example

In this example, a date of 2004-12-31 would return a value of 12 to Month:

```
#Month := #Today.Month
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.14 Now

Now returns the current date.

Input Parameters

None

Example

This example sets Today to the current date:

```
#Today := #Today.Now
```

[↑ 10.8 Date Intrinsic Functions](#)

10.8.15 Year

Year returns the year portion of a date.

Input Parameters

None

Example

In this example, a date of 2004-12-31 would return a value of 2004 to Year:

```
#Year:= #Today.Year
```

[↑ 10.8 Date Intrinsic Functions](#)

10.9 DateTime Intrinsic Functions

10.9.2 AsCustomDisplayString	10.9.5 AsSeconds	10.9.10 FractionalSeconds
10.9.1 AsDisplayString	10.9.6 AsString	10.9.11 IsNull
10.9.3 AsLocalizedDateTime	10.9.7 AsUniversalDateTime	10.9.12 IsSqlNull
10.9.4 AsNumber	10.9.8 AsValue	10.9.13 Now
	10.9.9 Date	10.9.14 Time

[↑ 10. Intrinsic Functions](#)

10.9.1 AsDisplayString

AsDisplayString returns the supplied date formatted using one of the available supplied date formats or a custom format.

Input Parameters

Supplied fixed formats and the resulting display string are listed below. Examples use a datetime of 2004-12-31 12:34:56.

The month is multilingual.

Format	Display String
CCYYDDMMHHMMSS	20043112123456
CCYYMMDDHHMMSS	20041231123456
HHMMSSbSysFmt6	123456 311204
HHMMSSbSysFmt8	123456 31122004
HHMMSSDDMMCCYY	12345631122004
HHMMSSDDMMYY	123456311204
LOCALIZED_SQL	2004-12-31 22:34:56.000000000
LOCALIZED_TZ	2004-12-31T22:34:56.000000000+10:00
SQL	2004-12-31 12:34:56.000000000
SysFmt6bHHMMSS	311204 123456
SysFmt8bHHMMSS	31122004 123456
TZ	2004-12-31T12:34:56.000000000Z

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.2 AsCustomDisplayString

AsCustomDisplayString returns the date and time formatted using the supplied formatting strings.

Input Parameters

Parameter 1: The date formatting string made up of a combination of the formats specified in CustomDateFormat. For details about valid date formats please see [CustomDateFormat](#).

Parameter 2: The time formatting string made up of a combination of the formats specified in CustomTimeFormat. For details about valid time formats please see [CustomTimeFormat](#).

Example

```
#STD_TEXT := #STD_DTIMX.AsCustomDisplayString( 'DD/MM/YY'  
'hh.mm.ss TT' )
```

Example output: 07/08/12 05.03.09 PM

```
#STD_TEXT := #STD_DTIMX.AsCustomDisplayString( 'DDDD, DD MMM  
YYYY' '(H:mm)' )
```

Example output: Wednesday, 07 Nov 2012 (17:03)

[↑ 10. Intrinsic Functions](#)

10.9.3 AsLocalizedDateTime

AsLocalizedDateTime returns the supplied datetime adjusted to the time zone specified on the executing system.

The time zone is the difference in time between the executing system and UTC (Universal Coordinated Time).

Input Parameters

None

Example

In this example, a date time of 2004-12-31 12:34:56 would return 2004-12-31 07:34:56 for US Eastern Standard Time (-5 Hours UTC):

```
#LocalTime := #DateTime.AsLocalizedDateTime
```

For more information about UTC, see [DateTime](#) field types.

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.4 AsNumber

AsNumber returns the supplied datetime, formatted as specified, as a number.

Input Parameters

Format - Required format of the datetime

Available formats, and the resulting numbers are listed below. Examples use a date of 2004-12-31.

Format	Display String
CCYYDDMMHHMMSS	20043112123456
CCYYMMDDHHMMSS	20041231123456
HHMMSSDDMMCCYY	12345631122004
HHMMSSDDMMYY	123456311204

Example

```
#Com_owner.Caption := #Today.AsNumber(CCYYDDMM)
```

See Input Parameters for the available formats and the resultant strings.

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.5 AsSeconds

AsSeconds returns the number of seconds since 00:00:00 for the specified date in the datetime

Input Parameters

None

Example

In this example, a date time of 2004-12-31 12:34:56 would return 45296:

```
#Seconds := #DateTime.AsSeconds
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.6 AsString

AsString returns the datetime as a string.

Input Parameters

None

Example

A datetime of 2004-31-12 12:34:56 will be returned as follows 2004-12-31
12:34:56:

```
#Com_owner.Caption := #DateTime.AsString
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.7 AsUniversalDateTime

AsUniversalDateTime returns the supplied datetime adjusted to UTC (Universal Coordinated Time) based on the time zone specified on the executing system.

The time zone is the difference in time between the executing system and UTC (Universal Coordinated Time).

Input Parameters

None

Example

```
#DateTime := #LocalTime.AsUniversalDateTime
```

In this example, a local date time of 2004-12-31 12:34:56 would return 2004-12-31 17:34:56 for US Eastern Standard Time (-5 Hours UTC)

For more information about UTC, see [DateTime](#) field types.

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.8 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable values for a date variable are any valid dates

Example

The IF below tests for the date as SQLnull or today's date

```
If ((#Datetime.IsSqlNull) *or (#Datetime = #Datetime.now))
```

```
Endif
```

Using AsValue it can be coded as follows

```
If (#Date.AsValue(#Datetime.now) = #Datetime.now)
```

```
Endif
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.9 Date

Date returns the date portion of the supplied datetime as a date.

Input Parameters

None

Example

In this example, a datetime of 2004-31-12 12:34:56 would return a date of 2004-31-12:

```
#Today := #DateTime.Date
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.10 FractionalSeconds

FractionalSeconds returns the decimal portion of a datetime as a number.

Input Parameters

None

Example

In this example, a date time of 2004-12-31 12:34:56.789 would return 789:

```
#Decimal := #DateTime.FractionalSeconds
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.11 IsNull

IsNull tests a datetime variable and returns a true if it contains a *null value.
The *null value for a datetime is 1900-01-01 00:00:00

Input Parameters

None

Example

```
#Button.enabled := #Datetime.IsNull
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.12 IsSqlNull

IsSqlNull tests a datetime variable and returns true if it contains an SQL Null

Input Parameters

None

Example

```
#Button.enabled := #Datetime.IsSqlNull
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.13 Now

Now returns the current datetime.

Input Parameters

None

Example

This example sets DateTime to the current date and time:

```
#DateTime := #DateTime.Now
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.9.14 Time

Time returns the Time portion of the supplied datetime as a time.

Input Parameters

None

Example

In this example, a datetime of 2004-31-12 12:34:56 would return a time of 12:34:56:

```
#Time := #DateTime.Time
```

[↑ 10.9 DateTime Intrinsic Functions](#)

10.10 Time Intrinsic Functions

In the following examples, simple names have been used to identify individual variables purely for the purposes of clarity. They are not representative of any variable in your LANSA repository.

[10.10.1 Adjust](#) [10.10.8 Difference](#)

[10.10.2 AsDateTime](#) [10.10.9 Hour](#)

[10.10.3 AsDisplayString](#) [10.10.10 IsNull](#)

[10.10.4 AsNumber](#) [10.10.11 IsSqlNull](#)

[10.10.5 AsSeconds](#) [10.10.12 Minute](#)

[10.10.6 AsString](#) [10.10.13 Now](#)

[10.10.7 AsValue](#) [10.10.14 Second](#)

[↑ 10. Intrinsic Functions](#)

10.10.1 Adjust

Adjust increments or decrements a time by the number of seconds specified

Input Parameters

Adjustment - Value to increment or decrement the time

Example

In this example, if #Time contained 12:34:56, the result would be 12:33:56:

```
#Later := #Time.Adjust( 60 )
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.2 AsDateTime

AsDateTime returns a DateTime variable that combines the Time and the Date specified in the Date parameter

Input Parameters

Date - Date to be combined with Time

Example

```
#DateTime := #Time.AsDateTime( #Date )
```

In this example, if #Time contained 12:34:56 and #Datetime contained 2004-12-31, the result would be 2004-12-31 12:34:56.

[↑ 10.10 Time Intrinsic Functions](#)

10.10.3 AsDisplayString

AsDisplayString returns the supplied Time formatted using one of the available supplied time formats

Input Parameters

Format - Required format of the data

Available formats, and the resulting display string are listed below. Examples use a time of 12:34:56

Format	Display String
HHMMSS	123456
HHsMMsSS	12:34:56
ISO	12:34:56

Example

```
#Com_owner.Caption := #Time.AsDisplayString( ISO )
```

See Input Parameters for the available formats and the resultant strings.

[↑ 10.10 Time Intrinsic Functions](#)

10.10.4 AsNumber

AsNumber returns the supplied Time converted to a number in one of the available supplied time formats.

Input Parameters

Format - Format of the returned time

Available formats are listed below. Examples use a time of 12:34:56

Format	Number
--------	--------

HHMMSS	123456
--------	--------

Seconds	45296
---------	-------

Example

```
#Seconds := #Time.AsNumber( Seconds )
```

See Input Parameters for the available formats and the resultant strings.

[↑ 10.10 Time Intrinsic Functions](#)

10.10.5 AsSeconds

AsNumber returns the supplied Time as the number of seconds since 00:00:00

Input Parameters

None

Example

In this example, if #Time had a value of 12:34:56, the result would be 45296:

```
#Seconds := #Time.AsSeconds
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.6 AsString

AsString returns the supplied Time as a string

Input Parameters

None

Example

```
#String := #Time.AsString
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.7 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable value for a time is any valid time.

Example

Rather than having to test as below

```
If ((#Time = 00:00:00) *or (#Time.IsSqlNull)
```

```
Endif
```

AsValue allows the following

```
If (#Time.AsValue( 00:00:00 ) = 00:00:00)
```

```
Endif
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.8 Difference

Difference returns the number of seconds between the subject and object times.

Input Parameters

Object - Time to be compared against

Example

In this example, if #Starttime contained 12:00:00 and #Endtime contained 13:00:00, the result would be 3600. That is, #Endtime is 3600 seconds (1 Hour) after #Starttime.

```
#Seconds := #EndTime.Difference(#StartTime)
```

Reversing the parameters, as follows, will give a result of -3600. That is, #starttime is 3600 seconds before #Endtime.

```
#Seconds := #StartTime.Difference( #EndTime )
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.9 Hour

Hour returns the Hour portion of a time.

Input Parameters

None

Example

In this example, if #Time contained 12:00:00, the result would be 12:

```
#Hour := #Time.Hour
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.10 IsNull

IsNull tests a time variable and returns a true if it contains a *null value.

Input Parameters

None

Example

The *null value for a time is 00:00:00:

```
#Button.enabled := #String.IsNull
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.11 IsSqlNull

IsSqlNull tests a time variable and returns true if it contains an SQL Null

Input Parameters

None

Example

```
#Button.enabled := #Time.IsSqlNull
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.12 Minute

Minute returns the minute portion of a time.

Input Parameters

None

Example

In the above Example, If #Time contained 12:34:00, the result would be 34:

`#Minute := #Time.Minute`

[↑ 10.10 Time Intrinsic Functions](#)

10.10.13 Now

Now returns the current time.

Input Parameters

None

Example

This Example sets time to the current time:

```
#Time := #Time.Now
```

[↑ 10.10 Time Intrinsic Functions](#)

10.10.14 Second

Second returns the seconds portion of a time.

Input Parameters

None.

Example

In this example, If #Time contained 12:34:56, the result would be 56:

```
#Seconds := #Time.Second
```

[↑ 10.10 Time Intrinsic Functions](#)

10.11 Large Object Intrinsic Functions

In the following examples, simple names have been used to identify individual variables purely for the purposes of clarity. They are not representative of any variable in your LANSAs repository.

[10.11.1 AsValue](#) [10.11.2 Filename](#) [10.11.3 IsNull](#)

[10.11.4 IsSqlNull](#)

[↑ 10. Intrinsic Functions](#)

10.11.1 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable value for a time is any valid time.

Example

Rather than having to test as below:

```
If ((#Blob = 'N/AVAILABLE') *or (#Blob.IsSqlNull))
```

```
Endif
```

AsValue allows the following:

```
If (#Blob.AsValue( 'N/AVAILABLE' ) = 'N/AVAILABLE')
```

```
Endif
```

[↑ 10.11 Large Object Intrinsic Functions](#)

10.11.2 Filename

FileName returns the qualified filename for the BLOB or CLOB. For BLOBs and CLOBs fields, this is equivalent to the field value

Input Parameters

None

Example

```
#FileName := #Blob.Filename
```

[↑ 10.11 Large Object Intrinsic Functions](#)

10.11.3 IsNull

IsNull tests a string variable and returns a true if it contains a *null value.

Input Parameters

None

Example

```
#Button.enabled := #String.IsNull
```

The *null value for a numeric variable is *blanks.

[↑ 10.11 Large Object Intrinsic Functions](#)

10.11.4 IsSqlNull

IsSqlNull tests a string variable and returns true if it contains an SQL Null.

Input Parameters

None

Example

```
#Button.enabled := #String.IsSqlNull
```

[↑ 10.11 Large Object Intrinsic Functions](#)

10.12 Binary Intrinsic Functions

In the following examples, simple names have been used to identify individual variables purely for the purposes of clarity. They are not representative of any variable in your LANSAs repository.

[10.12.1 AsByte](#) [10.12.5 IsNull](#)

[10.12.2 AsInteger](#) [10.12.6 IsSqlNull](#)

[10.12.3 AsString](#) [10.12.7 AsHexString](#)

[10.12.4 CurSize](#) [10.12.8 AsHexToInt](#)

10.12.1 AsByte

AsInteger returns the ASCII decimal value of the first character in the string, as per the table in [AsInteger](#) in String Intrinsic Functions.

Input Parameters

None

Example

Refer to the Example in [AsInteger](#) in String Intrinsic Functions.

[↑ 10.12 Binary Intrinsic Functions](#)

10.12.2 AsInteger

AsInteger returns the ASCII decimal value of the first character in the string, as per the table in [AsInteger](#) in String Intrinsic Functions.

Input Parameters

None

Example

Refer to the Example in [AsInteger](#) in String Intrinsic Functions.

[↑ 10.12 Binary Intrinsic Functions](#)

10.12.3 AsString

AsString is used to return a binary as a value of type string

Input Parameters

None

Example

```
#String := #Binary.AsString
```

[↑ 10.12 Binary Intrinsic Functions](#)

10.12.4 CurSize

CurSize is used to return the current byte length. Trailing spaces are significant.

Input Parameters

None

Example

In the example below, if #String contained the value "ABCDEFG", CurSize would return a value of 7

```
#StringLength := #String.CurSize
```

[↑ 10.12 Binary Intrinsic Functions](#)

10.12.5 IsNull

IsNull tests the binary and returns true if it contains a *Null (zero or blanks).

Input Parameters

None

Example

```
#Button.enabled := #Binary.IsNull
```

[↑ 10.12 Binary Intrinsic Functions](#)

10.12.6 IsSqlNull

IsSqlNull tests the binary and returns true if it contains an SQL Null

Input Parameters

None

Example

```
#Button.enabled := #Binary.IsSqlNull
```

[↑ 10.12 Binary Intrinsic Functions](#)

10.12.7 AsHexString

Convert integers to hexadecimal through Binary Strings.

A hexadecimal string cannot be represented in a numeric type, so the Binary String primitive is used as a staging area for the conversion.

Input Parameters

None

Portability Note that the byte order of the result depends on the byte order
Consideration of the computer being used.

Example

```
Define Field(#myRBStr) Type(*BIN) Length(128)
Define Field(#L8Int) Type(*INT) Length(8)
```

```
#myRBStr := (9999).AsBinString()
```

```
#myRBStr.AsHexString() gives 0F270000 (on Windows computers).
```

[↑ 10.12 Binary Intrinsic Functions](#)

10.12.8 AsHexToInt

Access the integer equivalent of a hexadecimal string using this intrinsic.

Input Parameters

None

Example

```
Define Field(#myRBStr) Type(*BIN) Length(128)
```

```
Define Field(#L8Int) Type(*INT) Length(8)
```

```
#L8Int := 9999
```

```
#L8Int := #L8Int.AsBinString().AsHexToInt()
```

```
#L8Int contains 9999.
```

[↑ 10.12 Binary Intrinsic Functions](#)

10.13 Decimal Intrinsic Functions

[10.13.1 AsValue](#)

[↑ 10. Intrinsic Functions](#)

10.13.1 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable values for a date variable are any valid dates

Example

The IF below tests for the date as SQLnull or less than 10:

```
If ((#Decimal.IsSqlNull) *or (#Decimal < 10))
```

```
Endif
```

Using AsValue it can be coded as follows

```
If (#Decimal.AsValue(0) < 10)
```

```
Endif
```

[↑ 10.13 Decimal Intrinsic Functions](#)

10.14 Fixed Point Intrinsic Functions

[10.14.1 AsFloat](#) [10.14.4 IsBetween](#) [10.14.7 Pred](#)
[10.14.2 AsString](#) [10.14.5 Max](#) [10.14.8 Round](#)
[10.14.3 Bound](#) [10.14.6 Min](#) [10.14.9 Succ](#)

[↑ 10. Intrinsic Functions](#)

10.14.1 AsFloat

AsFloat allows a fixed point number to be handled as a floating point number

Input Parameters

None

Example

```
#Float := #Number.Asfloat * #Float
```

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.2 AsString

AsString is used to return a number as a string.

Input Parameters

None

Example

```
#String := #number.AsString
```

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.3 Bound

Bound returns a number within the bounds in the supplied Input Parameters. If the variable is greater than the bounds, the upper bound is returned. If lower, the lower bound is returned. If the variable is within the bounds, the variable is returned unmodified

Input Parameters

Numberone - Upper or lower bound limit

Numbertwo - Upper or lower bound limit

Example

```
#Result := #Number.Bound(1 100)
```

Using the example above, if number contained 150, result would be set to 100. If number contained 0, result would be set to 1. If number contained 42, result would be set to 42.

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.4 IsBetween

IsBetween tests the value of a number and returns true if it is within the limits specified. A value equal to either of the limits is considered to be within the limits

Input Parameters

Numberone - Upper or lower limit

Numbertwo - Upper or lower limit

Example

```
#Button.enabled := #Number.IsBetween(1 100)
```

This is equivalent to writing

```
If ((#Number >= 1) and (#Number <= 100))
```

```
#Button.Enabled := True
```

```
Else
```

```
#Button.Enabled := False
```

```
Endif
```

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.5 Max

Max compares a numeric value to the value supplied in the Of parameter, and returns the larger of the two.

Input Parameters

Of - Numeric value to compare to

Example

```
#Result := #Number1.Max(#Number2)
```

In this example, if number1 has a value of 19, and number2 has a value of 20, result will be set to 20

This is equivalent to writing:

```
If (#number1 > #number 2)
```

```
#result := #number1
```

```
Else
```

```
#result := #number2
```

```
Endif
```

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.6 Min

Min compares a numeric value to the value supplied in the Of parameter, and returns the smaller of the two.

Input Parameters

Of - Numeric value to compare to

Example

In this example, if number1 has a value of 19, and number2 has a value of 20, result will be set to 19

```
#Result := Number1.Min(#Number2)
```

The above example is equivalent to writing

```
If (#number1 < #number 2)
```

```
#result := #number1
```

```
Else
```

```
#result := #number2
```

```
Endif
```

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.7 Pred

Pred returns the supplied variable decremented by 1.

Input Parameters

None

Example

```
If (#Previous = #Current.Pred)
```

This is equivalent to writing

```
If (#Previous = #Current - 1)
```

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.8 Round

Round allows a number to be rounded to a specified number of decimal places using a selected rounding technique.

Input Parameters

Operation - Type of rounding to be performed

Allowable values are

Up - will always round up

Down - will always round down

Halfup - will round up if the rounding value is 5 or more

Halfdown - will round up if the rounding value is 5 or less

See the Examples for more information on the behavior of each rounding type

Decimals - Number of decimal places to round to

Example

```
#Result := #Number.Round(Up 1)
```

Rounding up 10.51 to 1 decimal places will produce a result of 10.6.

Rounding up 10.01 to 0 decimal places will produce a result of 11.0.

Rounding down 10.51 to 1 decimal places will produce a result of 10.5.

Rounding up 10.99 to 0 decimal places will produce a result of 11.0.

Rounding halfup 10.49 to 1 decimal places will produce a result of 10.5.

Rounding halfup 10.44 to 1 decimal places will produce a result of 10.4.

Rounding halfdown 10.44 to 1 decimal places will produce a result of 10.4.

Rounding halfdown 10.46 to 1 decimal places will produce a result of 10.5.

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.14.9 Succ

Succ returns the supplied variable incremented by 1.

Input Parameters

None

Example

If (#Next = #Current.Succ)

This is equivalent to writing

If (#Next = #Current + 1)

[↑ 10.14 Fixed Point Intrinsic Functions](#)

10.15 Floating Point Intrinsic Functions

Note: The trigonometric intrinsics assume the input value is in radians, not degrees.

10.15.1 Add	10.15.12 Subtract	10.15.22 TangentH
10.15.2 AsDecimal	10.15.13 Succ	10.15.23 ArcTangent2
10.15.3 AsString	10.15.14 Sine	10.15.26 Power
10.15.4 AsValue	10.15.15 ArcSine	10.15.27 SQRT
10.15.5 Bound	10.15.16 SineH	10.15.28 Exponential
10.15.6 Divide	10.15.17 Cosine	10.15.29 Logarithm
10.15.7 IsBetween	10.15.18 ArcCosine	10.15.30 Logarithm10
10.15.8 Max	10.15.19 CosineH	10.15.24 FAbs
10.15.9 Min	10.15.20 Tangent	10.15.25 FMod
10.15.10 Multiply	10.15.21 ArcTangent	10.15.31 IsNANorND
10.15.11 Pred		

↑ 10. Intrinsic Functions

10.15.1 Add

Add adds the value specified in the Object parameter to the subject.

Input Parameters

Object – Value to be added to the subject

Example

```
#Float := #Float.Add(#Float2)
```

This is equivalent to writing

```
#Float += #Float2
```

or

```
#Float := #Float + #Float2
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.2 AsDecimal

AsDecimal allows a floating point number to a use fixed point number.

Input Parameters

None

Example

```
#Number := #Float1.asDecimal + #Float2.asDecimal
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.3 AsString

AsString is used to return a number as a string.

Input Parameters

None

Example

```
#String := #number.AsString
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.4 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable values for a date variable are any valid dates

Example

The IF below tests for the date as SQLnull or less than 10:

```
If ((#Number.IsSqlNull) *or (#Number < 10))
```

```
Endif
```

Using AsValue it can be coded as follows

```
If (#Number.AsValue(0) < 10)
```

```
Endif
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.5 Bound

Bound returns a number within the bounds in the supplied Input Parameters. If the variable is greater than the bounds, the upper bound is returned. If lower, the lower bound is returned. If the variable is within the bounds, the variable is returned unmodified

Input Parameters

Numberone - Upper or lower bound limit

Numbertwo - Upper or lower bound limit

Example

```
#Result := #Number.Bound(1 100)
```

Using the example above, if number contained 150, result would be set to 100. If number contained 0, result would be set to 1. If number contained 42, result would be set to 42.

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.6 Divide

Divide divides the subject by the value specified in the Object parameter.

Input Parameters

Object – Value by which the subject is divided

Example

```
#Float := #Float.Divide(#Float2)
```

This is equivalent to writing

```
#Float /= #Float2
```

or

```
#Float := #Float / #Float2
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.7 IsBetween

IsBetween tests the value of a number and returns true if it is within the limits specified. A value equal to either of the limits is considered to be within the limits

Input Parameters

Numberone - Upper or lower limit

Numbertwo - Upper or lower limit

Example

```
#Button.enabled := #Floating.IsBetween(1 100)
```

This is equivalent to writing

```
If ((#Floating >= 1) and (#Floating <= 100))
```

```
#Button.Enabled := True
```

```
Else
```

```
#Button.Enabled := False
```

```
Endif
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.8 Max

Max compares a numeric value to the value supplied in the Of parameter, and returns the larger of the two.

Input Parameters

Of - Numeric value to compare to

Example

```
#Result := #Number1.Max(#Number2)
```

In this example, if number1 has a value of 19, and number2 has a value of 20, result will be set to 20

This is equivalent to writing:

```
If (#number1 > #number 2)
```

```
#result := #number1
```

```
Else
```

```
#result := #number2
```

```
Endif
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.9 Min

Min compares a numeric value to the value supplied in the Of parameter, and returns the smaller of the two.

Input Parameters

Of - Numeric value to compare to

Example

In this example, if number1 has a value of 19, and number2 has a value of 20, result will be set to 19

```
#Result := Number1.Min(#Number2)
```

The above example is equivalent to writing

```
If (#number1 < #number 2)
```

```
#result := #number1
```

```
Else
```

```
#result := #number2
```

```
Endif
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.10 Multiply

Multiply multiplies the subject by the value specified in the Object parameter.

Input Parameters

Object – Value by which the subject is to be multiplied

Example

```
#Float := #Float.Multiply(#Float2)
```

This is equivalent to writing

```
#Float *= #Float2
```

or

```
#Float := #Float * #Float2
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.11 Pred

Pred returns the supplied variable decremented by 1.

Input Parameters

None

Example

```
If (#Previous = #Current.Pred)
```

This is equivalent to writing

```
If (#Previous = #Current - 1)
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.12 Subtract

Subtract subtracts the value specified in the Object parameter from the subject

Input Parameters

Object – Value by which the subject is subtracted

Example

```
#Float := #Float.subtract(#Float2)
```

This is equivalent to writing

```
#Float -= #Float2
```

or

```
#Float := #Float ;- #Float2
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.13 Succ

Succ returns the supplied variable incremented by 1.

Input Parameters

None

Example

If (#Next = #Current.Succ)

This is equivalent to writing

If (#Next = #Current + 1)

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.14 Sine

Applies the trigonometric sine() function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.234).AsFloat ().Sine()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.15 ArcSine

Applies the trigonometric arcsine() function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.570796327).AsFloat().ArcSine()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.16 SineH

Applies the trigonometric `sineh()` function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#afloat := (20.0).AsFloat().SineH()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.17 Cosine

Applies the trigonometric cosine() function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.2).AsFloat.Cosine()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.18 ArcCosine

Applies the trigonometric arccosine() function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.2).AsFloat.ArcCosine()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.19 CosineH

Applies the trigonometric `cosineh()` function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.2).AsFloat.CosineH()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.20 Tangent

Applies the trigonometric tangent() function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.2).AsFloat.Tangent()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.21 ArcTangent

Applies the trigonometric arctangent() function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.2).AsFloat.ArcTangent()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.22 TangentH

Applies the trigonometric `tangenth()` function to a float data type, returning a float result.

Input Parameters

None

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.2).AsFloat.TangentH()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.23 ArcTangent2

Applies the trigonometric arctangent2() function to a float data type, returning a float result.

Input Parameters

The "y" coordinate of the point.

Example

```
Define field(#aFloat) Type(*FLOAT)
#aFloat := (1.2).AsFloat().ArcTangent2(1.0)
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.24 FAbs

Applies the floating point fabs() function to a float data type, returning a float result that is the positive value of the float supplied.

Input Parameters

None

Example

```
(-9.0).AsFloat().FAbs()
```

yields the result 9.0

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.25 FMod

The FMod float intrinsic returns the modulus of the floating point number to which it is applied, as a floating point number.

Input Parameters

Modulus – this parameter defines the modulus

Example

```
#aFloat := (6.0).AsFloat().FMod( 4.0 )  
yields 2.0 in #aFloat
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.26 Power

Power() raises a floating point number to the power of the supplied parameter, returning a floating point number.

Input Parameters

PowerTo – raise the subject to this power

Example

```
(2.0).Power( 2.0 )
```

returns 4.0.

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.27 SQRT

Sqrt returns the square root of a floating point number, as a floating point number.

Input Parameters

None

Example

```
#aFloat := (9.0).Sqrt()  
places 3.0 in #aFloat
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.28 Exponential

Exponential() returns the exponential value of a given floating point number: e to the power of the number. The result is a floating point number.

Input Parameters

None

Example

```
#aFloat.Exponential()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.29 Logarithm

Returns the (natural) logarithm of a floating point number, as a floating point number.

Input Parameters

None

Example

```
#aFloat.Logarithm()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.30 Logarithm10

Returns the base 10 logarithm of a floating point number. The value is returned as a floating point number.

Input Parameters

None

Example

```
#aFloat.Logarithm10()
```

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.15.31 IsNaNorND

Certain intrinsics may return values that are not recognised numbers. An example is ArcCosine(3.0). The value returned may be either a NAN (not a number) or an ND(non-determinate).

IsNaNorNd() can be used to check a floating point number to determine if it is or is not a number. It returns a Boolean.

Input Parameters

None

Example

```
#afloat := (3.0).AsFloat().ArcCosine()  
#afloat.IsNaNorND()
```

yields true if the ArcCosine() returns a NAN.

[↑ 10.15 Floating Point Intrinsic Functions](#)

10.16 Integer Intrinsic Functions

[10.16.2 AsBinString](#)

[10.16.3 AsByte](#)

[10.16.4 AsChar](#)

[10.16.12 AsUnicodeString](#)

[10.16.5 AsValue](#)

[10.16.1 BinaryString Conversions](#)

[10.16.6 BitAnd](#)

[10.16.7 BitNot](#)

[10.16.8 BitOr](#)

[10.16.9 BitXOr](#)

[10.16.11 Div](#)

[10.16.10 Mod](#)

10.16.1 BinaryString Conversions

A hexadecimal string cannot be represented in a numeric type, so the BinaryString primitive is used as a staging area for the conversion.

Integers can be converted to and from hexadecimal strings by using the following intrinsics:

Given an integer, convert to binary string using `#integerValue.AsBinaryString()`,

Convert the binary string to a hexadecimal representation using `#binaryString.AsHexString()`,

Given a binary string containing a hexadecimal value, access as an integer value using `#binaryString.AsHexToInt()`.

These methods work with both integers and long integers.

For example:

```
Define Field(#myRBStr) Type(*BIN) Length(128) Define Field(#L8Int)
```

```
Type(*INT) Length(8) #myRBStr := (9999).AsBinString()
```

```
#myRBStr.AsHexString() gives 0F270000
```

```
#L8Int := 169999999999
```

```
L8Int.AsBinString().AsHexString() gives FF23CA9427000000 on intel-based computers.
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.2 AsBinString

AsBinString is used to create a binary string out of an integer value. This intrinsic can be used as a first step into creating a hexadecimal representation of an integer.

Input Parameters

None

Example

```
#STD_INT := 15
#STD_BIN := #STD_INT.AsBinString
#STD_TEXT := #STD_BIN.AsHexString
* #STD_TEXT will contain 0F000000 on intel-based computers
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.3 AsByte

AsByte returns the numeric code page value for the specified index

Input Parameters

None

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.4 AsChar

AsChar returns the subject as the equivalent character for the current code page

Input Parameters

None

Example

```
#String := "Start a new line" + (13).AsChar
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.5 AsValue

AsValue allows you to better handle *SQLNull. Rather than having to test for a potential SQL null, AsValue allows a variable to return a specified value.

Input Parameters

Default - The value to be returned when the variable contains an SQLNull.

Allowable values for a date variable are any valid dates

Example

The IF below tests for the date as SQLnull or less than 10:

```
If ((#Integer.IsSqlNull) *or (#Integer < 10))
```

```
Endif
```

Using AsValue it can be coded as follows

```
If (#Integer.AsValue(0) < 10)
```

```
Endif
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.6 BitAnd

BitAnd performs a logical bitwise and.

The BitXXX Intrinsic Functions are available in LANSAs to support the use of binary bit masks.

Input Parameters

None

Example

```
If (#Integer.BitAnd( 4 ) = 4)
```

```
    * Bit 4 is on
```

```
Else
```

```
    * Bit 4 is Off
```

```
Endif
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.7 BitNot

BitNot performs a logical bitwise not.

The BitXXX Intrinsic Functions are available in LANSAs to support the use of binary bit masks.

Input Parameters

None

Example

```
#Integer := #Integer.BitNot
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.8 BitOr

BitOr performs a logical bitwise or.

The BitXXX Intrinsic Functions are available in LANSAs to support the use of binary bit masks.

Input Parameters

None

Example

```
#Integer := #Integer.BitOr( 4 )
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.9 BitXOr

BitXOr performs a logical bitwise exclusive or (XOr).

The BitXXX Intrinsic Functions are available in LANSAs to support the use of binary bit masks.

Input Parameters

None

Example

```
#Integer := #Integer.BitXor( 4 )
```

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.10 Mod

Mod returns the modulus of an integer constant or field, when divided by another integer.

The result is an integer.

Input Parameters

By - divide by this number to obtain the modulus.

Example

```
#myIntField := 5
```

```
#myIntField.Mod( 3 )
```

yields the integer value 2.

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.11 Div

Div returns the number of times that an integer constant or field divides into another integer constant or field.

The result is an integer.

Input Parameters

By - divide by this number to obtain the result.

Example

```
#myIntField := 5
```

```
#myIntField.Div( 3 )
```

yields the integer value 1.

[↑ 10.16 Integer Intrinsic Functions](#)

10.16.12 AsUnicodeString

AsUnicodeString returns the subject code point as the equivalent Unicode character.

Input Parameters

None

Example

```
#My_Nvarchar := "Pythagoras in Greek: " + (928).AsUnicodeString +  
(965).AsUnicodeString + (952).AsUnicodeString + (945).AsUnicodeString +  
(947).AsUnicodeString + (972).AsUnicodeString + (961).AsUnicodeString +  
(945).AsUnicodeString + (962).AsUnicodeString
```

On a form, #My_Nvarchar would display Pythagoras in Greek: Πυθαγόρας

Also see

[10.17.1 AsCodePoint](#)

[↑ 10.16 Integer Intrinsic Functions](#)

10.17 Unicode String Functions

[10.17.2 AsNativeString](#)

[10.17.1 AsCodePoint](#)

10.17.1 AsCodePoint

AsCodePoint returns the first character of subject Unicode String as its numeric value or code point.

Input Parameters

None

Example

```
#STD_INT := #My_NVarchar.AsCodePoint
```

If #My_NVarchar contained 'A', #STD_INT would have the value 65.

Also see

[10.16.12 AsUnicodeString](#)

10.17.2 AsNativeString

AsNativeString returns the subject Unicode String converted to the current code page. Characters that are not valid in the current code page will be replaced with a question mark (?).

Input Parameters

None

Example

```
#STD_TEXTL := #My_NVarchar.AsNativeString
```

[↑ 10.17 Unicode String Functions](#)

[↑ 10.17 Unicode String Functions](#)

11. System and Multilingual Variables

A system variable is used to store commonly used pieces of information that are often variable or dynamic.

System variables are global, that is, system wide variables that are used across all LANSAs partitions.

To learn how to use LANSAs system variables, refer to [System Variable Concepts](#) in the *Developer Guide*.

LANSAs is shipped with a number of System Variables and they are listed in [11.2 Shipped System Variables](#)

Also See

[System Variable Evaluation Programs](#) in the *Visual LANSAs Developer Guide*.

To create system variables, refer to:

for IBM i: [Create a New System Variable](#) in the *LANSAs for i User Guide*.

for Visual LANSAs: [Edit System Variables](#) in the *Visual LANSAs User Guide*.

For the system variables details, refer to the [11.1 System Variable Definition](#).

11.1 System Variable Definition

Following are details of the information required for each system variable.

[11.1.1 Variable Name](#)

[11.1.2 Description](#)

[11.1.3 Derivation Method](#)

[11.1.4 Data Type](#)

[11.1.5 Length](#)

[11.1.6 Decimals](#)

[11.1.7 Program Type](#)

[11.1.8 Program Name](#)

Also See

[Edit System Variables](#) in the *Visual LANSa User Guide*.

[↑ 11. System and Multilingual Variables](#)

11.1.1 Variable Name

Mandatory.

Specify the name of the system variable to be stored in the repository.

For a list of shipped system variables, refer to [System and Multilingual Variables](#).

- Rules
- Must begin with an "*" (asterisk).
 - Must be at least 4 characters in length.
 - Must not contain imbedded blanks.
 - Must not begin with *MTXT as this prefix is reserved for multilingual variables.
 - Must not be *ALL, *ALL_REAL, *ALL_VIRT, *DEFAULT, *EXCLUDING, *HIVAL, *INCLUDING, *LOVAL, *NAVAIL, or *NULL as these values are reserved by LANSAP.

[↑ 11.1 System Variable Definition](#)

11.1.2 Description

Mandatory.

Specify a brief description of what the system variable is or represents, to aid other users of the system.

[↑ 11.1 System Variable Definition](#)

11.1.3 Derivation Method

Mandatory. Default= STATIC

Specifies how LANSAs is to derive the system variable within a file I/O module or user written function that references it. You will enter the program identifier in *Set Value by Calling* (on IBM i) and *Program Name* (in the LANSAs Editor).

Rules Allowable values are:

STATIC The system variable is a static value, therefore its value can be derived once (during program initialisation) by LANSAs. Examples of static system variables would include the current job name, the current user name and most probably the current date (providing that applications do not normally span midnight while executing).

DYNAMIC The system variable is a dynamic value, therefore its value must be derived each and every time it is referenced. Examples of dynamic system variables would include the current time, the current output queue name and library and all user defined system variables that "allocate" values such as the next invoice number, the next batch number, etc.

Note that every time a dynamic system variable is referenced the associated evaluation program is called to "refresh" the system variable. Excessive use of dynamic system variables with complex evaluation programs may degrade LANSAs performance.

[↑ 11.1 System Variable Definition](#)

11.1.4 Data Type

Mandatory. Default=ALPHA

Specify the field type of the system variable.

Rules Allowable values are:

ALPHA System variable is alphanumeric.

NUMBER System variable is numeric. Use of this option in fact nominates the system variable as a packed decimal variable.

[↑ 11.1 System Variable Definition](#)

11.1.5 Length

Mandatory.

Specify either the number of characters in an alpha system variable or the total number of digits (including decimals) in a numeric system variable.

- Rules
- Must be in range 1 to 256 for data type ALPHA.
 - Must be in range 1 to 30 for data type NUMBER.

[↑ 11.1 System Variable Definition](#)

11.1.6 Decimals

Optional.

Specify the number of decimals for a numeric type system variable.

- Rules
- Must be in range 0 to 9 and less than or equal to total digits.
 - Must be entered for data type NUMBER.
 - Ignored for data type ALPHA.

[↑ 11.1 System Variable Definition](#)

11.1.7 Program Type

Mandatory.

Specify either the name of the LANSAs function or the 3GL program that is to be called to set the value of the system variable.

Specify if a LANSAs function is to be called to set the value of the system variable.

Specify if a 3GL program is to be called to set the value of the system variable.

[↑ 11.1 System Variable Definition](#)

11.1.8 Program Name

Mandatory.

Specify the name of the LANSAs function or 3GL program that is to be called to set the value of the system variable.

- Rules
- LANSAs function name must not exceed 7 characters.
 - Program names may be 10 characters.

- Warnings
- LANSAs checks that the 3GL program or function name specified is valid, but does not check that it actually exists.
 - The program should be able to be located at the time the system variable is evaluated.

- Platform Considerations
- IBM i: The program should be able to be located in the user's library list at the time the system variable is evaluated.

For details about system variable evaluation programs, refer to [System Variable Evaluation Programs](#) in the *Visual LANSAs Developer Guide*.

[↑ 11.1 System Variable Definition](#)

11.2 Shipped System Variables

It is strongly recommended that you do not change the shipped system variables or delete them from the system.

For your convenience, they are listed in these groups:

[11.2.1 General Variables](#)

[11.2.2 Authenticating User System Variables](#)

[11.2.3 Function Only Variables](#)

[11.2.4 Special Variables](#)

[11.2.5 SuperServer System Variables](#)

[11.2.6 Built-In Function Variables](#)

[↑ 11.1 System Variable Definition](#)

11.2.1 General Variables

These system variables are supplied in the shipped version of LANSAs. Do not change these system variables or delete them from the system.

System Variable	Description
*AT_CHAR	The "@" character
*BLANK	Blank/blanks variable
*BLANKS	Blank/blanks variable
*CENTURY_GREATER	Century when date greater than switch
*CENTURY_LESSEQUAL	Century when date less/equal to switch
*CENTURY_SWITCH	Century compare date
*CHECKBOXSELECTED	Selected check box value
*COMPANY	Name of current company/organization
*COMPILECPU	<p>The CPU that the runtime is compiled for. This variable is provided for completeness. Consider *OSAPI or *OSBITNESS before using this one to make your program more portable. For example, you may be targeting a Tablet and Windows Desktop and so you could differentiate between them using *COMPILECPU, but it may be better to use *OSAPI as behaviour is likely to be common across all WINRT devices and all Windows Desktop devices, no matter which CPU those devices are using.</p> <p>Intel x86 and Intel x86-x64 chips – INTELX (Intel Itanium (IA-64) is NOT an environment that LANSAs supports) ARM chips - ARM Power chips - POWER</p>
*COMPONENT	Name of the active component when reference in a component context (i.e. in or from RDML)

logic) or equivalent to *FUNCTION when referenced in a non-component context

*CPFREL	Current OS/400 or CPF version level
*CPU_NUMBER	CPU Serial Number. On IBM i platforms, a valid value is only returned in RDML applications.
*CPUTYPE	CPU type. This variable is for backwards compatibility only. Use *OSAPI and/or *OSBITNESS instead. When used, AS/400, iSeries and IBM i all return the value AS400. Linux returns the value UNIX Windows returns the value WINNT.
*DATE	Numeric date in installation format. Refer to the Date Note .
*DATE8	Numeric 8 digit date in installation format Refer to the Date Note .
*DATE8C	Character 8 digit date in installation format Refer to the Date Note .
*DATEC	Character date in installation format. Refer to the Date Note .
*DATETIME	Current date and time (numeric) Refer to the Date Note .
*DATETIMEC	Current date and time (character) Refer to the Date Note .
*DAY	Current day (numeric) Refer to the Date Note .
*DAYC	Current day (character) Refer to the Date Note
*DDMMYY	Numeric date in format DDMMYY Refer to the Date Note .
*DDMMYYC	Character date in format DDMMYY Refer to the Date Note .

*DDMMYYYY	Numeric date in format DDMMYYYY Refer to the Date Note .
*DDMMYYYYC	Character date in format DDMMYYYY Refer to the Date Note .
*DEVELOPMENTLANGUAGE	Development Language For LANSA internal use only.
*DOLLAR_CHAR	The "\$" character
*FIELD_PREFIX	Field prefix
*FUNCTION	Current LANSA function name or name of active component when referenced in component context, that is, in or from RDMLX logic.
*GROUP_AUTHORITY	Group profile authority
*GROUP_OWNER	Group profile owner
*GROUP_PROFILE	Group profile
*GUID	Globally Unique Identifier . Usually incorporated into a File using the field STD_GUID.
*GUIDEVICE	GUI device in use (Y=GUI, N=NPT)
*JOBMODE	Current job mode (B=batch, I=inter)
*JOBNAME	Current IBM i job name
*JOBNBR	Current IBM i job number
*JULIAN	Numeric date in Julian format Refer to the Date Note .
*JULIANC	Character date in Julian format Refer to the Date Note .
*LANGUAGE	Current language code
*LANGUAGE_DESC	Current language description

*LANGUAGE_IGC	Current language is IGC/DBCS (Y/N)
*LANGUAGE_LRTB	Current language is Left to Right (Y/N)
*LANGUAGE_RLTB	Current language is Right to Left (Y/N)
*LANSACOMLIB	LANSA communication library
*LANSADTALIB	LANSA system data/file library
*LANSAPGMLIB	LANSA system program library
*LASTFUNCTION	Last LANSA function name
*MESSAGE_FILE	Message file name
*MMDDYY	Numeric date in format MMDDYY. Refer to the Date Note .
*MMDDYYC	Character date in format MMDDYY. Refer to the Date Note .
*MMDDYYYY	Numeric date in format MMDDYYYY. Refer to the Date Note .
*MMDDYYYYC	Character date in format MMDDYYYY. Refer to the Date Note .
*MONTH	Current month (numeric). Refer to the Date Note .
*MONTHC	Current month (character), Refer to the Date Note .
*MSGQLIB	Current message queue library
*MSGQNAME	Current message queue name
*NEXTFUNCTION	Default next LANSA function name
*ON_CLIENT_SYSTEM	On IBM i running an RDML function, value is always N. In all other situations, including an RDMLX function on an IBM i, Y indicates that the currently executing LANSA object has direct access to a user interface, otherwise this value

N.

Note:

When running a LANSAs object from a DB2 Trigger on IBM i the value is Y. Use of *CPUYPE = AS400 can be used to distinguish this situation.

*ON_SERVER_SYSTEM

On IBM i running an RDML function, value is always Y.

In all other situations, including an RDMLX function on IBM i, Y indicates that this LANSAs object was executed through a server interface such as SuperServer or LANSAs for the Web, otherwise this value is N. Further, Y indicates that the currently executing LANSAs object does NOT have direct access to a user interface.

Note:

When running a LANSAs object from a DB2 Trigger on IBM i, the value is N. Use of *CPUYPE = AS400 can be used to distinguish this situation.

*ORGANISATION

Name of current company/organization

*OSAPI

Operating System API Name.

This is a more precise replacement for *CPUYPE which is less likely to change the name of its values.

Windows Desktop - WIN32 (Note Windows Desktop 64-bit uses WIN32 API)

Windows Metro – WINRT

IBM i – IBMI

Linux - LINUX

*OSBITNESS

Operating System Bitness.

For comparing the behaviour of the operating system that may differ between, say, 32-bit and 64-bit applications.

For example, to access different registry hives

when using a 32-bit application as opposed to a 64-bit application on Windows. Or, to load a DLL which only exists in 32-bit.

32-bit operating system - 32

64-bit operating system – 64 (Note that current IBM i is 64-bit - only pointers are 128)

- *OUTQLIB Current output queue library name
- *OUTQNAME Current output queue name
- *PART_DIR The root directory of the current partition's system. For example, for Windows 32-bit:
D:\X_WIN95\X_LANSA\X_DEM\
For Windows 64-bit:
D:\X_WIN64\X_LANSA\X_DEM\
On IBM i platforms, a valid value is only returned in RDMLX applications.
- *PART_DIR_EXECUTE The directory of the current partition's EXECUTE objects. For example, for Windows 32-bit:
D:\X_WIN95\X_LANSA\X_DEM\EXECUTE\
For Windows 64-bit:
D:\X_WIN64\X_LANSA\X_DEM\EXECUTE\
On IBM i platforms, a valid value is only returned in RDMLX applications.
- *PART_DIR_OBJECT The directory of the current partition's OBJECT objects. For example, for Windows 32-bit:
D:\X_WIN95\X_LANSA\X_DEM\OBJECT\
For Windows 64-bit:
D:\X_WIN64\X_LANSA\X_DEM\OBJECT\
On IBM i platforms, a valid value is only returned in RDMLX applications.
- *PART_DIR_SOURCE The directory of the current partition's SOURCE objects. For example, for Windows 32-bit:
D:\X_WIN95\X_LANSA\X_DEM\SOURCE\
For Windows 64-bit:
D:\X_WIN64\X_LANSA\X_DEM\SOURCE\
On IBM i platforms, a valid value is only returned in RDMLX applications.

	On IBM i platforms, a valid value is only returned in RDMLX applications.
*PART_DRIV	This should not be used as it does not support UNC naming. The drive of the current partition's LANSAs system. A driver letter followed by a colon. For example: C: or D: or E: This is for backward compatibility. On IBM i platforms, a valid value is only returned in RDMLX applications.
*PART_RDMLX	Y if current partition is RDMLX enabled, otherwise N.
*PARTDTALIB	Current partition's data/file library
*PARTITION	Current partition
*PARTITION_DESC	Current partition description
*PARTPGMLIB	Current partition's RDML pgm library
*PATHDELIM	'\ ' if running on an MS Windows system '/' if running on a Linux system. On IBM i platforms, a valid value is only returned in RDMLX applications.
*PROCESS	Current LANSAs process name or name of active component when referenced in component context, that is, in or from RDMLX logic.
*PRODREL	Current LANSAs version level
*PRODUCT	Product name (i.e.: LANSAs)
*QUOTE	Quote character (i.e. ')
*RADBUTTONSELECTED	Selected Radio Button Value
*ROOT_DIR	LANSAs Root directory
*SYS_DIR	The root directory in which the LANSAs system

is located.

For example, for Windows 32-bit:

D:\X_WIN95\X_LANSA\

For Windows 64-bit:

D:\X_WIN64\X_LANSA\

On IBM i platforms, a valid value is only returned in RDMLX applications.

*SYS_DIR_EXECUTE

The directory of the LANSAs system's EXECUTE objects. For example, for Windows 32-bit: D:\X_WIN95\X_LANSA\EXECUTE\
For Windows 64-bit:
D:\X_WIN64\X_LANSA\EXECUTE\
On IBM i platforms, a valid value is only returned in RDMLX applications.

*SYS_DIR_OBJECT

The directory of the LANSAs system OBJECT objects. For example, for Windows 32-bit: D:\X_WIN95\X_LANSA\OBJECT\
For Windows 64-bit:
D:\X_WIN64\X_LANSA\OBJECT\
On IBM i platforms, a valid value is only returned in RDMLX applications.

*SYS_DIR_SOURCE

The directory of the LANSAs system SOURCE objects. For example, for Windows 32-bit: D:\X_WIN95\X_LANSA\SOURCE\
For Windows 64-bit:
D:\X_WIN64\X_LANSA\SOURCE\
On IBM i platforms, a valid value is only returned in RDMLX applications.

*SYS_DRIV

For backward compatibility. Should not be used as it does not support UNC naming. The drive of the LANSAs system. A drive letter followed by a colon. For example: C: or D: or E:.
On IBM i platforms, a valid value is only returned in RDMLX applications.

*TEMP_DIR

The directory in which temporary files / objects

should be created e.g.: D:\TEMP\

*TEMP_DRIV	For backward compatibility. Should not be use as it does not support UNC naming. The drive in which temporary files/objects should be created. For example: C: or D: or E: (A driver letter followed by a colon.) On IBM i platforms, a valid value is only returned in RDMLX applications.
*TIME	Current time (numeric)
*TIMEC	Current time (character)
*TIMEDATE	Current time and date (numeric)
*TIMEDATE8	Current time and 8 digit date (numeric)
*TIMEDATE8C	Current time and 8 digit date (character)
*TIMEDATEC	Current time and date (character)
*TIMESTAMP_DFT	Timestamp default value
*TIMESTAMP_HIVAL	Timestamp high value
*TIMESTAMP_LOVAL	Timestamp low value
*VISUAL_LANSA	Y if executing on a Visual LANSA system, otherwise 'N'
*WEBIPADDR	IP address of the current user.
*WEBMODE	Y indicates that LANSA application is running under web-enabled mode.
*WEBPATHINFO	PATH_INFO Web server environment variable
*WEBREFERRER	HTTP_REFERER Web server environment variable
*WEBSCRIPTNAME	SCRIPT_NAME Web server environment variable
*WEBUSER	Web Server/400 or Internet Connection Server for IBM i user profile

*XMLMODE	Y indicates that LANSAs application is running as XML/Java Thin Client
*YEAR	Current year (numeric). Refer to the Date Note
*YEARC	Current year (character). Refer to the Date Note ..
*YYMMDD	Numeric date in format YYMMDD. Refer to the Date Note .
*YYMMDDC	Character date in format YYMMDD. Refer to the Date Note .
*YYYY	Numeric year in format YYYY. Refer to the Date Note .
*YYYYC	Character year in format YYYY. Refer to the Date Note .
*YYYYMMDD	Numeric date in format YYYYMMDD. Refer to the Date Note .
*YYYYMMDDC	Character date in format YYYYMMDD. Refer to the Date Note .
*ZERO	Zero (0) variable
*ZEROES	Zero (0) variable
*ZEROS	Zero (0) variable

Date Note

For any LANSAs-supplied system variable that contains or is derived from date, the following is true:

The date is derived from either the system value QDATE or the job attribute date, depending on the DATE_SRCE value of the LANSAs command, at the point in time that the LANSAs environment was entered. This means that it does not change when the system value QDATE changes or the job date is changed. Applications requiring this feature should use a user-defined dynamic system variable.

For information about creating and using System Variables, refer to:

- [Create System Variables](#) in the *Visual LANSAs User Guide*.
- [Creating A New System Variable](#) in the *LANSAs for i User Guide*.

Also See

[System Variable Evaluation Programs](#) in the *Visual LANSAs Developer Guide*.

[↑ 11. System and Multilingual Variables](#)

***GUID System Variable**

To generate a Globally Unique Identifier, use the *GUID system variable.

*GUID uses a Mersenne twister [Pseudorandom number generator](#) (PRNG) as a GUID. It has no dependency on any machine state – for example, it is not dependent on the MAC address or current time. It is also thread safe. The same algorithm is used on Windows, IBM i and Linux.

The Mersenne twister PRNG provides fast generation of very high-quality pseudorandom integers; it was designed specifically to rectify many of the flaws found in older PRNGs. Its name derives from the fact that its period length is chosen to be a [Mersenne prime](#). LANSAs implements MT19937. See this link for more information: [Mersenne_twister](#). **Note:** If sufficient numbers are observed (624) it is possible to predict all future iterations, so it is not suitable for cryptographic purposes.

*GUID is expected to be generated once when inserting a row and then not to be updated again. A standard field, #STD_GUID, has been provided to implement this behaviour. This field has a Trigger. Before Insert it assigns *GUID to the field. Before Update it ensures the field has not changed, and if it has changed it ABORTs as it is a programming level error, not a user error.

11.2.2 Authenticating User System Variables

These are the variables described below:

System Variable	Description	Type	Len	Dec
*USER	Current IBM i User ID	A	10	
*USER_AUDIT	Current Audit User Identity. Refer to SET_SESSION_VALUE and Output Stamping Attributes .	A	256	
*USER_AUTHENTICATED	Current authenticated user name.	A	256	
*WEBUSER	Web Server/400 or Internet Connection Server for IBM i user profile	A	10	

***USER**

*USER represents the User ID with which the Visual LANSAs application (x_run) was started. If running a function through LANSAs SuperServer, *USER represents the User ID used to connect to the server. If running in a Web Environment, *USER stores the LANSAs User ID. Mappings between the User ID used to logon to the web session and LANSAs User ID's are created through the Web Administrator.

***USER_AUTHENTICATED**

*USER_AUTHENTICATED represents the User ID which has been used to log on to the operating system and which has been authenticated by the selected authentication protocol (such as Kerberos). If running a function through LANSAs SuperServer, *USER_AUTHENTICATED represents the User ID used to connect to the server. The authenticated name is the fully qualified name which includes the User ID and the domain/workgroup/machine name. For example, JohnCitizen@mydomain.com.au, JohnCitizen@Johnspc.

In LANSAs SuperServer mode, if the User ID is defined both as a local user and a domain user, the local user will be used for authentication.

In the web runtime environment, *USER_AUTHENTICATED represents the

User ID used to log on to the web session.

- If *Integrated Windows Authentication* is specified, the User ID used to log on to the operating system is used for the web session, otherwise
- the User ID specified when the session is started is used.
- If *Anonymous Access* is specified, *USER_AUTHENTICATED is blank.

Please refer to [*WEBUSER](#) for more details.

Examples:

Start Form As	John	
Windows log in	MYDOMAIN\John	
Connect As	John	
Connect To	Windows (Logged in as MYDOMAIN\John on server)	
	USER	USER_AUTHENTICATED
Local	John	John@MYDOMAIN.COM.AU
Server side	John	John@MYDOMAIN.COM.AU

Start Form As	John	
Windows log in	MYDOMAIN\John	
Connect As	John	
Connect To	IBM i	
	USER	USER_AUTHENTICATED
Local	John	John@MYDOMAIN.COM.AU
Server side	John	

Start Form As	John	
Windows log in	MYPC\Mary	
Connect As	John	
Connect To	Windows (Logged in as MYDOMAIN\John on server)	
	USER	USER_AUTHENTICATED
Local	John	mary@mypc
Server side	John	John@MYDOMAIN.COM.AU

Start Form As	John	
Windows log in	MYPC\Mary	
Connect As	Mary	
Connect To	Windows (Logged in as MYPC\Mary on server)	
	USER	USER_AUTHENTICATED
Local	John	John@MYDOMAIN.COM.AU
Server side	John	mary@serverpc

***WEBUSER**

*WEBUSER represents the User ID used to log on to the web session. If

Integrated Windows Authentication is specified, the User ID used to log on to the operating system is used for the web session. Otherwise the User ID specified when the session is started is used. The User ID will be truncated to the length of *WEBUSER.

*WEBUSER is the first ten bytes of the *USER_AUTHENTICATED value.

Examples:

With the following User Registration settings:

Web User ID	Web Server Name	Web Server Port	LANSA User ID	Timeout
John	<ANY>	<ANY>	luser1	0
DFTUSR	<ANY>	<ANY>	luser2	0
A123456789B123456789	<ANY>	<ANY>	luser3	0

Windows log in	MYDOMAIN\John
Connect As	Integrated Windows Authentication
Connect To	Windows server
*WEBUSER	John
*USER	luser1
*USER_AUTHENTICATED	John

Windows log in	MYDOMAIN\Mary
Connect As	Integrated Windows Authentication
Connect To	Windows server
*WEBUSER	Mary
*USER	luser2

*USER_AUTHENTICATED	Mary
----------------------------	------

Windows log in	MYDOMAIN\John
Connect As	Basic Authentication (login: mydomain\Mary)
Connect To	Windows server
*WEBUSER	Mary
*USER	luser2
*USER_AUTHENTICATED	Mary

Windows log in	MYDOMAIN\John
Connect As	Basic Authentication (login: A123456789B123456789)
Connect To	IBM i
*WEBUSER	A123456789
*USER	luser3
*USER_AUTHENTICATED	A123456789B123456789

↑ 11. System and Multilingual Variables

11.2.3 Function Only Variables

These system variables can only be used in RDML functions and RDMLX logic. They have no meaning in any other part of the system (that is, they cannot be used as the default on a field defined in the repository). If you use these variables in another context, this value will be returned:

"## XXXXXXXX description not available ##"

System Variable	Description	Type	Length	Dec
*COMPONENT_DESC	Description of the active component when referenced in or from RDMLX logic. When referenced in a non-component context, this value is equivalent to *FUNCTION_DESC.	A	40	
*FUNCTION_DESC	Function or component description (Function use only)	A	40	
*PROCESS_DESC	Process or component description (Function use only)	A	40	

Note: System variables *FUNCTION_DESC and *PROCESS_DESC are supplied in the current language and centered on IBM i and left aligned in Windows.

Also See

[System Variable Evaluation Programs](#) in the *Visual LANSAs Developer Guide*.

[11.2.6 Built-In Function Variables](#)

[↑ 11. System and Multilingual Variables](#)

11.2.4 Special Variables

Remember: System variables exist at the LANSAsystem level and are shared by all partitions. When the value of a system variable is incremented by an application in one LANSAsystem partition, then all partitions will now use the new value.

[System Variables for use in System Evaluation Programs](#)

[System Variables to test I/O Status](#)

[↑ 11. System and Multilingual Variables](#)

System Variables for use in System Evaluation Programs

The following system variables can be used with the system evaluation programs that have been shipped with this product.

On a SuperServer client with locks diverted to the server, these variables are retrieved from the server. Refer to [DEFINE_ANY_SERVER](#) for details.

System Variable	Description
*DTAssslllxxxxxxxxx	Special data area system variable layout when used in conjunction with evaluation program M@SYSDTA. Retrieves data at position sss for a length ll from data area xxxxxxxxxx. Length is limited to 256 characters.
*AUTOALPnnxxxxxxxxx	Special data area system variable layout when used in conjunction with evaluation program M@SYSNUM. Retrieves a number nn long from data area xxxxxxxxxx. Increments it, updates data area and returns as an alphanumeric value. Length is limited to 15 digits.
*AUTONUMnnxxxxxxxxx	Special data area system variable layout when used in conjunction with evaluation program M@SYSNUM. Retrieves a number nn long from data area xxxxxxxxxx. Increments it, updates data area and returns as a numeric value. Length is limited to 15 digits.

For details about the evaluation programs, refer to [System Variable Evaluation Programs](#) in the *Visual LANSAs Developer Guide*.

[↑ 11.2.4 Special Variables](#)

System Variables to test I/O Status

The following system variable can be used in functions to check for record locked I/O error. See [I/O Status Record Locked](#) for details of using this system variable.

On a SuperServer client when the I/O request has been run on the server, *DBMS_RECORD_LOCKED retrieves the server record lock details. Refer to [DEFINE_ANY_SERVER](#) for details.

System Variable	Description	Type	Len	Dec
*DBMS_RECORD_LOCKED	I/O status record locked	Y/N A	1	

[11.2.4 Special Variables](#)

11.2.5 SuperServer System Variables

System Variable	Description
*SSERVER_CONNECTED	Indicates (as Y or N) whether a current SuperServer connection exists.
*SSERVER_SSN	Returns current SuperServer connection SSN (Symbolic Server Name).
*SSERVER_TYPE	Returns current SuperServer connection type as one of: AS400 - returned when you connect to an IBM i server with DEFINE_OS_400_SERVER (or DBID=*OS400) RDMLX400 - returned when connect to an IBM i server with DEFINE_ANY_SERVER (or DBID=*ANY) OTHER - returned when you connect to a Windows or Linux server NONE - returned when you are not connected to a server.

[↑ 11. System and Multilingual Variables](#)

11.2.6 Built-In Function Variables

These system variables can only be used in Built-In Functions. They have no meaning in any other part of the system (That is, they cannot be used as the default on a field defined in the repository). If you use these variables in another context, this value will be returned:

```
"## XXXXXXXX description not available ##"
```

System Variable	Description	Type	Len	Dec
*BIF_SHUTDOWN	Shutdown call (Y, N)	A	1	
*BIF_ARGCOUNT	Number of arguments	N	7	0
*BIF_RETCOUNT	Number of return values	N	7	0

[↑ 11. System and Multilingual Variables](#)

11.3 Multilingual Text Variables

A multilingual variable is a text string that changes value according to the language being used. It is a special form of system variable that is specific to a multilingual partition. Note that multilingual text variables are not system wide as are system variables.

To create or edit Multilingual Variables refer to [Multilingual Variables](#) in the *Visual LANSAs User Guide*.

For details about using them refer to [Multilingual Variables](#) in the *Multilingual Application Design Guide*.

[11.3.1 MTXT Variable Name](#)

[11.3.2 Maximum Length](#)

[11.3.3 Value](#)

[↑ 11. System and Multilingual Variables](#)

11.3.1 MTXT Variable Name

Mandatory.

Specify the name of the multilingual variable.

Rules

- Must use "*MTXT" for first 5 characters.
- Rest of the name must be from 1 to 15 characters and cannot contain imbedded blanks.
- Must be unique within the current partition.

Warnings

- Most *MTXT references are included at compile time. Change of an *MTXT variable will require application recompile. This is where the field/default value/cross reference capability is most useful. Exception is use of *MTXT variables as message text. In this situation the derivation is dynamic, so no recompiles are required.

Tips & Techniques

- Use a maximum of 3 characters for function key names as the input field on RDML commands is only 8 characters long.
- Develop and use naming standards for *MTXT variables.
- In packaged systems, use obscure prefixes to preserve uniqueness.
- Use as default value for fields in the repository, rather than direct RDML reference, is preferable when intention is to use as panel or report text. Improves cross referencing capabilities.
- Fields in the repository should have a naming standard too. It should equate to the naming standard used for the multilingual variable that is used as its default value.

[↑ 11.3 Multilingual Text Variables](#)

11.3.2 Maximum Length

Mandatory.

Specify the maximum length of multilingual variable.

Rules

- Must be within range 1 to 78.

Warnings

- No [11.3.3 Value](#) entered should exceed length specified.

Tips & Techniques

- In RLTB languages, length is from the right hand side.
- Includes shift characters in DBCS languages.

Note: If a value greater than 78 is entered a length of 78 is assumed.

[↑ 11.3 Multilingual Text Variables](#)

11.3.3 Value

Mandatory. Specify the value that the *MTXT variable is to have when the specified language is being used.

Rules

- Must not exceed [11.3.2 Maximum Length](#) specified.

Tips & Techniques

- Keyed from right in RLTB languages.
- Include shift characters in DBCS languages.
- Use upper and lower case characters as required.
- Manually centering (centering) within the maximum length can be used.
- Centre from left (within maximum length) for LRTB and DBCS languages.
- Centre from right (within maximum length) for RLTB languages.
- If no specific value has been entered for a language, the value for the default partition language will be used for that language.

Also See

[↑ 11.3 Multilingual Text Variables](#)

12. Formats, Values and Codes

12.1 LANSAs Object Names

12.2 Date Formats

12.3 Standard Field Edit Codes

12.4 RDML Field Attributes and their Use

12.5 RDML I/O Return Codes

12.6 Help Text Enhancement & Substitution Values

12.1 LANSa Object Names

It is strongly recommended that you review all object naming rules.

Object Long Name, Short Name and Identifier Relationship

Each LANSa object has two names – a long name and an identifier. Generally, either name may be used to refer to a LANSa object.

An RDML partition only permits identifiers to be used. An RDMLX partition may be enabled for long names.

Short Name is another term for an identifier. The LANSa Editor has a setting to show short names. With this set, all Editor browser windows will display the identifier for the object name.

Where a LANSa Guide uses the term "object name", either a long name or an identifier may be used, unless otherwise specified.

General Name Rules

Following are some general rules and guidelines that apply to ALL objects stored in the LANSa Repository:

- LANSa object names must be unique in a LANSa partition. For example, if a field is created with the name EMPNO, then no other object (file, component, function, etc.) can use this name.
- Long names must not be an existing identifier and vice versa. This allows either to be used anywhere an object name is required as these are unique names in the partition.
- Names are NOT case sensitive. The EMPNO and Empno and EmpNo are considered the same name.
- Embedded blank characters are not allowed in names. For example, "EMP NO" is not a valid name, however, "EMP_NO" is a valid name for some objects.
- It is recommended that you do not use these reserved prefixes and reserved names: _, X, X_, COM, COM_, SYS, SYS_, SYSTEM, SYSTEM_, LAN, LAN_, LANSa, LANSa_ and PRIM_ in your object names.

Object Long Name Rules

- Long names must only contain the characters a-z, A-Z, and 0 – 9.
- The case of the letter as entered is retained for easier readability, but it must be unique insensitive to case. E.g. a long name of Aa is stored and displayed

everywhere as Aa, but another object cannot be called AA or aa.

- A long name may be up to 256 characters long.
- The first 128 characters must be unique in a LANSAs partition and must not be the same as an identifier.
- All LANSAs object types have the same rules, apart from Field Long Names and File Long Names which have implementation differences depending on the target databases required as described below.

Field Long Name Rules

- Oracle and DB2 for i have a maximum column identifier length of 30 characters. If either of these databases is targeted and any Field long name in the LANSAs file is longer than 30 characters ALL columns will use the LANSAs field identifier (short name) in ALL databases.

File Long Name Rules

- If the physical file or logical view has a Long Name that is less than 9 characters, the identifier will be used instead.
- Oracle has a maximum table identifier length of 30 characters. If Oracle is targeted and the File long name is longer than 30 characters the table identifier will use the LANSAs file identifier (short name) in ALL databases.
- Note that the table identifier restriction is only on Oracle. All the other supported databases use 128 bytes.

Object Identifier (Short Name) Rules

- LANSAs converts all identifiers to uppercase characters in the repository.
- First character of names should be A to Z. (Characters \$, @ and # are allowed in some names but are not recommended.)
- For simplicity, it is strongly recommended that you use only characters A to Z and 1 to 9 in LANSAs object names. Using special characters (#, _, @, \$, etc.) are allowed in some object names but may have portability and other impacts.

Field Identifiers

- First character must be A to Z. Do not use @ anywhere in field identifiers. Characters \$, _, and # are allowed but are not recommended.
- Field identifiers are restricted to a maximum of 9 characters.
- Avoid the use of field identifiers like SQLxxx, as this may cause problems when used in functions that use SQL (Structured Query Language) facilities. (IE Command SELECT_SQL.)

Component identifiers

- First character must be A to Z. Do not use @ anywhere in component identifiers. Characters \$, _, and # are allowed but are not recommended.
- Component identifiers are restricted to a maximum of 9 characters.

File Identifiers

The following rules apply to both physical and logical file identifiers:

- File identifiers must be valid for the target operating system and DBMS.
- File identifiers are restricted to a maximum of 10 characters.
- The first character must be A to Z or \$, #, @. Remaining characters may be A to Z, 0 to 9, or \$, #, @. The use of "_" (underscore) is not allowed.
- Access route are considered part of a file definition and are not considered a separate LANSAs object. Their names must be unique within the file definition. Access route names must follow general naming rules.

Process Identifiers

- The process identifier must be unique within the entire LANSAs System.
- A maximum of 8 character process identifiers is recommended. Maximum length for a process identifier is 10 characters.
- IBM i: A process identifier must be unique within a LANSAs partition. A function identifier must be unique within the process it is created.
- Windows: A process identifier must be unique within the entire LANSAs system. All functions must be defined as type *DIRECT. Functions identifier must be unique in the partition.
- Windows: If 10 character process identifier are used on Windows, the last 9 characters must be unique as first character is truncated when generating some program names.

Function Identifiers

- RDML function identifier must not use "_" (underscore).
- Function identifier Fnnnnnn/Cnnnnnn/Pnnnnnn (where nnnnnn is in range 1 to 999999) are reserved words.
- Function identifier MENU, EXIT, HELP, SELECT, EOJ, ERROR, RETRN, and *ANY are reserved.
- Function identifier are restricted to a maximum of 7 characters.
- IBM i: A function identifier must be unique within the process it is created.

- Windows: All functions must be defined as type *DIRECT. Function identifiers must be unique in the partition.

Platform Considerations

- Characters such as #, \$ or @ may have language code translation issues if your application is executing in more than one country. Using special characters with caution or simply avoid using special characters in object identifiers.
- For multiplatform applications, LANSAs object identifiers should only contain the characters A to Z or 0 to 9 because these characters do not change between the different code pages. Thus when using communications between operating systems on different platforms, the object identifiers will match correctly.

12.2 Date Formats

Date Built-In Functions may use one or more of the following formats as arguments and/or return values. The relevant Built-In Function's description tells you which format is used as arguments and/or return values.

Code Format of Date

- A System date format
- B DDMMYY
- C DD/MM/YY
- D YYMMDD
- E YY/MM/DD
- F MMDDYY
- G MM/DD/YY
- H DDMMYYYY
- I DD/MM/YYYY
- J YYYYMMDD
- K YYYY/MM/DD
- L MMDDYYYY
- M MM/DD/YYYY
- N DDMMYY (e.g. 03JUL87)
- O DDMMYYYY (e.g. 03JUL1987)
- P DDxx MMMMMMMMMM YY (e.g. 16TH SEPTEMBER 87)
- Q DDxx MMMMMMMMMM YYYY (e.g. 16TH SEPTEMBER 1987)
- R DDD (e.g. MON, TUE, WED, etc).
- S DDDDDDDDD (e.g. MONDAY, TUESDAY, WEDNESDAY, etc).
- T DDDDDDDDD (in selected language e.g. LLLLLLLLLL).

U MMMMMMMMMM (in selected language e.g. LLLLLLLLLL).

V 8 digit system date format

W YYMM

X MMY

Y YYYYMM

Z MMY

1 CYMMDD

12.3 Standard Field Edit Codes

Field Edit Codes are stored in the Repository and are used to return your data in the format required by the application.

If your output is not in the format expected, you may simply need to change the edit code in order to fix it.

	Code	Commas	Decimal Point	Leading Zero Suppression	Zero Balance	Type of Sign used	Position
	1	Y	Y	Y	Y	None	
	2	Y	Y	Y		None	
	3		Y	Y	Y	None	
	4		Y	Y		None	
	A	Y	Y	Y	Y	CR	Trailing
	B	Y	Y	Y		CR	Trailing
	C		Y	Y	Y	CR	Trailing
	D		Y	Y		CR	Trailing
	J	Y	Y	Y	Y	-	Trailing
	K	Y	Y	Y		-	Trailing
	L		Y	Y	Y	-	Trailing
	M		Y	Y		-	Trailing
	N	Y	Y	Y	Y	-	Leading
	O	Y	Y	Y		-	Leading
	P		Y	Y	Y	-	Leading
	Q		Y	Y		-	Leading
	W **						
	Y *			Y			Y
	Z *			Y			

** Note that *W* is a special date edit code with i5/OS version considerations. Refer to the IBM manual *Data Description Specifications*, keyword DDS, RPG/400 and/or *ILE RPG for AS400 Reference Edit Codes* section.

* Note that *Y* is a special date edit code and *Z* is a special sign removal edit code.

12.4 RDML Field Attributes and their Use

Whenever fields are declared in a FIELDS parameter they can have various attributes associated with them. This applies equally to the FIELDS parameter of an I/O command such as FETCH, DISPLAY or UPRINT and to the FIELDS parameter of a GROUP_BY, DEF_LIST, DEF_HEAD, DEF_FOOT, DEF_LINE or DEF_BREAK command.

Attributes assigned to fields in expandable groups are ignored. Refer to [Special Considerations for Expandable Groups](#) for details.

When a field in a FIELDS parameter is to have attributes associated with it, it must be individually enclosed in parenthesis with its attributes. For instance consider the following, where a1 ... a7 are the special attributes assigned to field #ORDLIN:

```
REQUEST  FIELDS(#ORDLIN #PRODUCT)
```

or the identical commands:

```
GROUP_BY #ORDERLINE FIELDS(#ORDLIN #PRODUCT)
REQUEST  FIELDS(#ORDERLINE)
```

If the field #ORDLIN is to be assigned some special attributes, then the commands would have to be modified like this:

```
REQUEST  FIELDS((#ORDLIN a1 a2 a3 a4 a5 a6 a7) #PRODUCT)
GROUP_BY #ORDERLINE FIELDS((#ORDLIN a1 a2 a3 a4 a5 a6 a7)
                        #PRODUCT)
REQUEST  FIELDS(#ORDERLINE)
```

Up to 7 special attributes may be assigned to any field in a list or group.

Attribute Notes

Attributes assigned to fields in expandable groups are ignored.

In this example, the attributes of field FA001 in the expandable group #XG_001 are ignored in the REQUEST command:

```
GROUP_BY NAME(#XG_001) FIELDS((#FA001 *BLUE *BL) (#FA002))
GROUP_BY NAME(#XG_002) FIELDS((#XG_001) (#FA003))
REQUEST  FIELDS(#XG_002)
```

Attributes assigned to expandable groups within a list or another expandable group are also ignored.

In this example, the attributes assigned to the expandable group #XG_001 in the REQUEST command are ignored:

```
GROUP_BY NAME(#XG_001) FIELDS((#FA001) (#FA002) (#FA003))
REQUEST FIELDS((#XG_001 *BLUE *BL))
```

Attributes assigned to individual fields in a field list, which include expandable group entries are acknowledged.

In this example, only the attributes assigned to field #FA005 in the REQUEST command are acknowledged:

```
GROUP_BY NAME(#XG_001) FIELDS((#FA001 *BLUE *BL) (#FA002))
GROUP_BY NAME(#XG_002) FIELDS((#FA003 *BL) (#FA004))
REQUEST FIELDS((#XG_001) (#XG_002) (#FA005 *BLUE *BL))
```

Refer to these topics for a list and examples of special attributes that can be used with a field:

[12.4.1 Output Only Attributes](#)

[12.4.2 Field Conditioning Attributes](#)

[12.4.3 Field Display Attributes](#)

[12.4.4 Field Identification Attributes](#)

[12.4.5 Field Position Attributes](#)

[12.4.6 Hidden Field Attribute and the Select Field Attribute](#)

[12.4.7 New Format Attribute and Repeat Attributes](#)

[12.4.8 Print Control Attributes](#)

12.4.1 Output Only Attributes

The following attributes are synonyms. Use of these attributes indicates that the field is an "output only" or a "no change" field and it should ALWAYS be protected from user change when it is displayed on a screen.

- *NC
- *NOCHG
- *NOCHANGE
- *OUT
- *OUTPUT
- *OUTONLY

Example:

This command indicates that fields #A, #B and #C should be displayed to the user and the CHANGE function key should be enabled (which will make the screen input capable and allow change of information on the screen). However, field #C has attribute *NOCHG, which indicates that it should not be allowed to be changed:

```
DISPLAY FIELDS(#A #B (#C *NOCHG)) CHANGE_KEY(*YES)
```

The following attributes are synonyms. Use of these attributes indicates that the field is an "input field" and it should NEVER be protected from change, no matter what the screen processing mode is at the time. Refer to the following sections for more information about screen modes. Input Only Attributes

- *IN
- *INP
- *INPUT

Example:

This command indicates that fields #A, #B and #C should be displayed to the user. If the screen is in "display" mode fields #A and #B will be protected from user change. However, field #C has attribute *INPUT, which indicates that it should always be "input capable" no matter what the screen mode:

```
DISPLAY FIELDS(#A #B (#C *INPUT))
```

12.4.2 Field Conditioning Attributes

Attributes	Description
*axxxxxxxxx	<p>Where *axxxxxxxxx is the name of a condition previously defined by a DEF_COND command. The name *axxxxxxxxx must conform to the naming standard that applies to defining conditions. Using a condition name that conflicts with another type of attribute name is not advisable (e.g.: *UL, *IN, etc)</p> <p>Use of an *axxxxxxxxx attribute alone indicates that the associated field should only appear on the screen panel or report when the condition is true.</p>
*INOUTCOND	These attributes are synonyms.
*IOCOND	<p>Use of these attributes with the *axxxxxxxxx attribute specifies that it is not the presence or absence of the field on the screen panel that is to be conditioned, but rather whether or not the field is protected from input or change by the user.</p> <p>If the associated condition is true, then the field will be input capable (regardless of the current screen mode). Similarly, if the associated condition is not true then the field will be protected from input (regardless of the current screen mode).</p> <p>If you use more than one *IOCOND (or *INOUTCOND) as a field attribute, only the last one will be used to control whether the field is input capable or not. Any others are ignored.</p>
	Important notes
	:
	<p>This attribute is not valid in report layouts. Additionally, this attribute was implemented for special situations encountered by some users. If you find that you are using it continuously, on many fields on many panels, then you should seriously consider simplifying the architecture of your application to reduce the usage of this attribute.</p>

Example:

This command cause a display panel to be created so that the field #SALARY only **appears** on the screen panel when the department number is 462 (accounting department) or the application group is HOFF (Head Office):

```
DEF_COND NAME(*AUTSAL) COND('#DEPT = 462) *OR (#GROUP = HOFF)
DISPLAY FIELDS(#A #B (#SALARY *AUTSAL) #C #D #E #F)
```

However, the commands cause a display panel to be created so that the field #SALARY is **input capable** when the department number is 462 (accounting department) or the application group is HOFF (Head Office). In all other cases field SALARY will appear, but it will be protected from input (ie: change) by the user:

```
DEF_COND NAME(*CHGSAL) COND('#DEPT = 462) *OR (#GROUP = HOFF)
DISPLAY FIELDS(#A #B (#SALARY *CHGSAL *IOCOND) #C #D #E #F)
```

12.4.3 Field Display Attributes

Attributes	Description
*AB	Allow to be blank.
*ME	Mandatory entry check required.
*MF	Mandatory fill check required.
*M10	Modulus 10 check required.
*M11	Modulus 11 check required.
*VN	Valid name check required.
*FE	Field exit key required.
*LC	Lowercase entry allowed. If this attribute is NOT specified, refer to <i>PC Locale uppercasing requested</i> in Review or Change a Partition's Multilingual Attributes in the <i>LANSA for i User Guide</i> .
*LCASE	
*LOWER	
*LOWERCASE	
*RB	Right adjust and blank fill.
*RZ	Right adjust and zero fill.
*RL	Move cursor right to left.
*RLTB	Tab cursor right/left top/bottom. Valid in SAA/CUA partitions only. Affects all screen panels in function.
*GRN	Display with color green.
*GREEN	
*WHT	Display with color white.
*WHITE	
*RED	Display with color red.

*TRQ	Display with color turquoise.
*TURQ	
*YLW	Display with color yellow.
*YELLOW	
*PNK	Display with color pink.
*PINK	
*BLU	Display with color blue.
*BLUE	
*BL	Display blinking.
*CS	Display with column separators.
*HI	Display in high intensity.
*ND	Non-display (hidden field).
*RA	Auto record advance field
*SREV	Store in reversed format. This special attribute is provided for bi-directional languages and is not applicable in this context.
SBIN	Store in binary format. This special attribute is provided for repository fields & is not applicable in this context.

Example:

Display field #A in blue, #B in green and allow field #C to be entered in lowercase characters:

```
DISPLAY FIELDS((#A *BLUE)(#B *GRN)(#C *LC))
```

12.4.4 Field Identification Attributes

Attributes	Description
------------	-------------

*COLUMN	These attributes are synonyms.
---------	--------------------------------

*COL	Indicates that the field should be identified on the screen by its column headings.
*COLHEAD	
*COLHDG	

*LAB	These attributes are synonyms.
------	--------------------------------

*LABEL	Indicates that the field should be identified on the screen by its label.
--------	---

*DES	These attributes are synonyms.
------	--------------------------------

*DESC	Indicates that the field should be identified on the screen by its description.
-------	---

This attribute is only valid in SAA/CUA compliant partitions. Additionally, when fields on a screen panel use attribute *DES/*DESC directly, or by default, they are automatically padded with "leader dots" ending with a "." (input field) or a ":" (protected field). The maximum length of

all
descriptions will be the maximum length of the longest description of

any
field on the screen panel plus 6 characters (for "... " or "... :"). Attributes *DES/*DESC

cannot
be used for reports.

*NOID	These attributes are synonyms.
-------	--------------------------------

*NOIDENT	Indicates that the field should not be identified on the screen. Only the field is to appear.
----------	---

Examples

This command specifies that field #A is to be identified by its column headings, field #B is not to be identified and fields #C, #D and #E are to be identified by their respective labels (because the IDENTIFY parameter nominates the default identification method for fields that do not have a specific identification attribute):

```
DISPLAY FIELDS((#A *COL)
(#B *NOID) #C #D #E) IDENTIFY(*LABEL)
```

This command specifies that all fields except for #E are to be identified by their column headings. Field #E is not to be identified. Only the field itself is to appear on the screen. :

```
DISPLAY FIELDS(#A #B #C #D (#E *NOID)) IDENTIFY(*COLHDG)
```

12.4.5 Field Position Attributes

Attributes Description

- *Rnnn These attributes are synonyms.
- *Lnnn These attributes are used to indicate a specific row / line on the screen / report at which the field should be positioned.
Note: These are ignored if specified for a field in a browselist.
- *Cnnn These attributes are synonyms.
- *Pnnn These attributes are used to indicate a specific column / position on the screen / report at which the field should be positioned.

Special note

(right-to-left languages only): If you are using a right-to-left language, and have not specifically disabled the automatic "mirroring" facility (see the FUNCTION command), the column / position specified will be automatically "inverted" or "mirrored" into a right-to-left panel or report position. This facility provides for the easy change of panel and report layouts from right- to-left into left-to-right layouts, or vice-versa. However it does mean that all positions are

always

specified

in left-to-right format. Thus it is

much

easier to modify panel and report layouts by using the screen or report painter facilities than by making manual RDML changes.

Example

This command specifies, for a left-to-right language, that field #A, prefixed by its data dictionary label, should be positioned at row 2, position 10 and that field #B should be positioned at row 5, position 15:

```
DISPLAY FIELDS((#A *L2 *P10)(#B *L5 *P15))
```

Note that the manual specification of row and column numbers for fields can be an arduous task. It is much quicker and easier to use the LANSAs screen design facility, which will automatically generate the required row and column numbers.

For Visual LANSAs, refer to [Function Screen Designer](#) in the *Visual LANSAs User Guide* for details.

For IBM i, refer to the [The Screen Design Facility](#) in the *LANSAs for i User Guide*.

12.4.6 Hidden Field Attribute and the Select Field Attribute

Attributes Description

- *HIDE These attributes are synonyms.
- *HIDDEN Use of these attributes indicates that the field is to be "hidden" and not displayed on the screen.
This attribute is primarily intended to allow fields to be included into a browse list but not actually displayed on the screen. Refer to the DEF_LIST command for more information about lists and list processing.
- *SEL These attributes are synonyms.
- *SELECT Use of this attribute indicates that a field is to be used to "select" an entry from a list. Fields with this attribute are input capable no matter what the display mode. Refer to the DEF_LIST and SELECTLIST command for more details of lists and list processing.

Examples

The following RDML program uses the *HIDDEN and *SELECT attributes and requests that the user input a generic customer name. All customer names that match are displayed and any of them can be selected for detailed display:

```
DEFINE FIELD(#CHOOSE) TYPE(*CHAR) LENGTH(1) COLHDG('Sel')
DEF_LIST NAMED(#BROWSE) FIELDS((#CHOOSE *SELECT) #NAME
GROUP_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDR1
REQUEST FIELDS(#NAME)
CLR_LIST NAMED(#BROWSE)
SELECT FIELDS(#BROWSE) FROM_FILE(CUSMSTV1) WITH_KEY(#N.
ADD_ENTRY TO_LIST(#BROWSE)
ENDSELECT

DISPLAY BROWSELIST(#BROWSE)

SELECTLIST NAMED(#BROWSE) GET_ENTRIES(*SELECT)
```

```

FETCH   FIELDS(#CUSTOMER) FROM_FILE(CUSMST) WITH_KEY(#C
DISPLAY FIELDS(#CUSTOMER)
ENDSELECT

```

Some points to note about this RDML program are:

- The first block of executable commands requests that the user input a customer name and then builds a list of all customers that have a generically identical name.
- The first DISPLAY command displays the list built by the first block of code. When displayed the list would look something like this:

```

Sel   Customer name
-   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
-   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
-   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

Note the input capable "Sel" column. This resulted from assigning the *SELECT attribute to field #CHOOSE.

Notice also that field #CUSTNO does not appear on the display. This is because it has attribute *HIDDEN. Even though it is not on the display it is still part of each list entry and is used in the final loop to fetch the required customer record for detailed display.

- The SELECTLIST / ENDSELECT loop causes entries in the list to be processed. The GET_ENTRYYS(*SELECT) parameter indicates that only entries which have a non-blank value in field #CHOOSE should be selected for processing. Note the link between the GET_ENTRYYS parameter and the *SELECT attribute in the DEF_LIST command.
- Within the SELECTLIST / ENDSELECT loop the field #CUSTNO which was defined as being in the list (but hidden from the user) is used to retrieve the correct customer record for the detailed display. This had to be done in this case because the field #NAME does not provide a unique key for the customer involved.

The *SELECT attribute can also be used in various other ways. Consider the following example:

```

DEFINE FIELD(#CHOOSE) TYPE(*CHAR) LENGTH(3) COLHDG('Se
DEF_LIST NAMED(#BROWSE) FIELDS((#CHOOSE *SELECT) #ORDE

```

```

REQUEST  FIELDS(#DATDUE)
CLR_LIST NAMED(#BROWSE)
SELECT  FIELDS(#BROWSE) FROM_FILE(ORDHDRV3) WITH_KEY
ADD_ENTRY TO_LIST(#BROWSE)
ENDSELECT

```

```

DISPLAY  BROWSELIST(#BROWSE)
--> SELECTLIST NAMED(#BROWSE) GET_ENTRYS(*SELECT)
|
|  CASE    OF_FIELD(#CHOOSE)
|
|  WHEN    VALUE_IS('= CUS')
|          < display customer details >
|  WHEN    VALUE_IS('= DET' '= LIN')
|          < display line item details >
|  WHEN    VALUE_IS('= HIS')
|          < display customer payment history >
|  WHEN    VALUE_IS('= STS')
|          < display order status >
|
|  ENDCASE
|
--- ENDSELECT

```

- Note that like the first example the commands request that a "date order due" be input. A list of all associated orders is then built and displayed. When displayed the list would look something like this:

Sel	Order	Date Due
_	9999999	99/99/99
_	9999999	99/99/99
_	9999999	99/99/99
_	9999999	99/99/99

- The SELECTLIST / ENDSELECT loop requests that all entries in the list that have a non-blank value in field #CHOOSE be processed. However, the loop also acts upon the content of field #CHOOSE to display customer, line item, payment history or status information about the order.

12.4.7 New Format Attribute and Repeat Attributes

Attributes	Description
*NEWFORMAT	This attribute is valid in DISPLAY, REQUEST or POP_UP commands and indicates that the field should be placed on a new screen format. Note: This attribute is ignored by Visual LANSA.
*REPEAT	This attribute is valid in DISPLAY, REQUEST or POP_UP commands and indicates that the field should be repeated onto each and every format required.

Examples

This command indicates that the DISPLAY command should design 2 separate screen formats. The first should include fields #A #B #C. The second (which was triggered by the *NEWFORMAT attribute) should contain fields #D, #E and #F:

```
DISPLAY FIELDS(#A #B #C (#D *NEWFORMAT) #E #F)
```

When a DISPLAY, REQUEST or POP_UP command uses multiple screen formats (either because all the fields specified will not fit on one format or because a *NEWFORMAT attribute is used) it can be treated like one "long" format. When the DISPLAY or REQUEST command is executed all resulting formats will be displayed in order before the next RDML command is executed. Refer to the DISPLAY or REQUEST command for more details.

This command indicates that fields #A -> #Z should be displayed. In addition field #A has the attribute *REPEAT. This indicates to LANSA that if fields #A -> #Z will not fit on one format, then field #A should be repeated on each and every additional format that is designed:

```
DISPLAY FIELDS((#A *REPEAT) #B #C ..... #Z)
```

12.4.8 Print Control Attributes

Print control attributes apply mostly to the UPRINT (unformatted print) command. The PRINT command (and associated DEF_XXXXX commands) facilitates all features described here in different and more advanced ways. Refer to the PRINT and DEF_XXXXX print definition commands for more details.

Also refer to [Producing Reports Using LANSAs](#) in the *Visual LANSAs Developer Guide*.

The use of the UPRINT command is recommended only for very simple list style reports. For serious application reporting, multilingual reporting or bi-directional language reporting use **only** the PRINT command.

Attribute	Applies To UPRINT	Applies To PRINT	Description
*NEWPAGE	YES	NO	Indicates that a change of the contents of this field should cause a new page to be started.
*TOTAL *TOT	YES	NO	Indicates that this field should be subtotaled.
*TOTLVLn *TOTLEVELn	YES	NO	Indicates that a change to the contents of this field should trigger a subtotal to be printed at "level break" n.
*ONCHANGE	YES	NO	Indicates that this field should only be printed when its contents changes.
*NOPRINT *NOPRT	YES	NO	Indicates that this field should not be printed.

Example

Consider a file called ACCOUNTS that contains the following fields and data:

Company	Division	Department	Expenditure	Revenue
(#COMP)	(#DIV)	(#DEPT)	(#EXPEND)	(#REVNU)

01	1	ADM	400	576
"	"	MKT	678	56
"	"	SAL	123	6784
"	2	ADM	46	52
"	"	SAL	978	456
"	3	ACC	456	678
"	"	SAL	123	679
02	1	ACC	843	400
"	"	MKT	23	0
"	"	SAL	876	10
"	2	ACC	0	43

If the file is keyed by #COMP, #DIV and #DEPT, the following RDML program will produce a paginated report with subtotals from this file:

```
GROUP_BY NAME(#ACCOUNTS) FIELDS((#COMP *TOTLEVEL1
    *NEWPAGE *ONCHANGE)
    (#DIV *TOTLEVEL2)
    (#DEPT *TOTLEVEL3)
    (#EXPEND *TOTAL)
    (#REVNU *TOTAL))
```

```
SELECT FIELDS(#ACCOUNTS) FROM_FILE(ACCOUNT)
UPRINT FIELDS(#ACCOUNTS)
ENDSELECT
```

```
ENDPRINT
```

The following points about the field attributes used should be noted:

- The *NEWPAGE attribute indicates that a new page should be started whenever the company number changes.
- The *ONCHANGE attribute indicates that field #COMPNO should only ever be printed when it changes.

- The *TOTLEVELn attributes indicate the total "level breaks" that are required. In this case, totals are required by company, division (within company), and department (within division within company).
- The *TOTAL attribute indicates the fields that are to totaled. In this case the expenditure and revenue fields are to be totaled.
- Grand totaling is implicit. Once the *TOTAL attribute is used a grand total line will be automatically produced when the print file is closed (see the ENDPRINT command).

12.5 RDML I/O Return Codes

Most LANSAs database commands issue a "return code" when they have completed. This return code is always mapped into a field called #IO\$STS which can be used in conditional statements like any other field. Optionally the return code can also be mapped into a user defined field. Refer to the IO_STATUS parameter of the required command for more information about how this is done.

However, the approach which should be taken in all normal commercial functions is that if there was a fatal error, allow the automatic error handler to take care of it. Either the I/O operation worked, or it didn't work (and if it didn't the messages will explain why, not the return code).

The list of all I/O return code values and their meanings are as follows:

Return Description / Meaning Code

- | | |
|----|--|
| OK | OKAY. Operation completed normally. No errors detected. |
| ER | FATAL ERROR. Fatal file error detected. Error is probably irrecoverable. Locate cause of problem, correct, and re-attempt the operation.

See also the section in this chapter that describes locked I/O status records. |
| VE | VALIDATION ERROR. Insert, update or delete operation failed to satisfy a file or dictionary level validation check. |
| NR | NO RECORD. No record(s) could be found matching the request. |
| EF | END OF FILE. End of file detected during read operation. |
| BF | BEGINNING OF FILE. Beginning of file detected during a read backwards. |
| EQ | EQUAL KEY FOUND. A record with a key equal to the key specified was found in the file. |
| NE | NO EQUAL KEY FOUND. No record could be found with a key equal to the key specified. |

There are various ways of checking the return code after an I/O operation has been performed.

The **first** is to always use the IO_STATUS(*STATUS) default parameter on an I/O command. In this case the return code is mapped into a field called #IO\$STS which can be referenced just like any other field. For example:

```
FETCH FIELDS(#ORDERHEAD) FROM_FILE(ORDHDR) WITH_KEY(#
IF COND('#IO$STS *NE OK')
MESSAGE MSGTXT('Order not found in current order file')
ENDIF
```

The **second** is to use the IO_STATUS parameter to map the return code into a user defined field. For example:

```
DEFINE FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)
FETCH FIELDS(#ORDERHEAD) FROM_FILE(ORDHDR) WITH_KEY(#
IO_STATUS(#RETCODE)
IF COND('#RETCODE *NE OK')
MESSAGE MSGTXT('Order not found in current order file')
ENDIF
```

The **third**, and probably the best, is to use the IF_STATUS command to test the last return code automatically. The example already used would become:

```
FETCH FIELDS(#ORDERHEAD) FROM_FILE(ORDHDR) WITH_KEY(
IF_STATUS IS_NOT(*OKAY)
MESSAGE MSGTXT('Order not found in current order file')
ENDIF
```

Refer to the [IF_STATUS](#) command for more details and examples.

Also see

[I/O Command Return Codes Table](#)

12.6 Help Text Enhancement & Substitution Values

Help text can be input for fields, processes and functions.

For Visual LANSA, how to add or amend Help text is described in [Field Help Text](#) and [Process Help Text](#).

For IBM I, how to add or amend Help text is described in the [Field Help Text](#) and [Process Help Text](#) in the *LANSA for i User Guide*.

You can include actual values when the HELP text is displayed by inserting the special substitution/control values in these lists:

[12.6.1 Substitution/Control Values](#)

[12.6.2 Substitution/Control Values - Visual LANSA only](#)

[12.6.3 Help Text Attributes](#)

When creating new HELP text for a field, process or function an option is available that allows you to create help using a "standard form".

The "standard form" allows the HELP text associated with any field, process or function to be formatted to a standard layout, thus making all HELP text input by users consistent in layout and format.

12.6.1 Substitution/Control Values

Substitution Value	Description, Effects And Comments
\$\$PAGE	Causes the string "Page : 999" to be inserted into the text. Use to indicate the current HELP text page number to the user.
\$\$NEWPAGE	Causes a new HELP text page to be started. A HELP text page is 18 lines long by default. The line containing the \$\$NEWPAGE value is included into the displayed HELP text. The \$\$NEWPAGE value is replaced by blanks.
\$\$TITLE XXXXXXXX	Indicates to LANSAs the title that should be associated with the HELP text. Up to 40 characters of title can be specified. LANSAs will automatically center the title and convert it to uppercase. The line containing the \$\$TITLE value is NOT included into the displayed help text.
\$\$FLDNAM	Causes the name of the current field to be inserted into the HELP text. Use this value in field level HELP text only.
\$\$FLDDES	Causes the description of the current field to be inserted into the HELP text. Use this value in field level HELP text only.
\$\$PRONAM	Causes the name of the current process to be inserted into the HELP text. Use this value in process or function level HELP text only.
\$\$PRODES	Causes the description of the current process to be inserted into the HELP text. Use this value in process or function level HELP text only.
\$\$FUNNAM	Causes the name of the current function to be inserted into the HELP text. Use this value in function level HELP text only.
\$\$FUNDES	Causes the description of the current function to be

inserted into the HELP text. Use this value in function level HELP text only.

\$\$RIGHT

Causes all manually defined help text to be right aligned when displayed. This support is provided for bi-directional languages.

\$\$NOAUTO

Indicates that automatically generated help text should not be created for this field. Use this option in field level help text only.

\$\$LANGUAGE=XXXX

Delimits the boundary between the help text associated with different languages when working in a multilingual partition. These values are

automatically created

when editing help text and

should not be altered

in any way or the help text associated with a language may be corrupted.

\$\$TECH

Indicates that the following text is technical help text in IBM i

\$\$USER

Indicates that the following text is user help text in IBM i.

Also See

[How to Use Special Characters](#) in the *Visual LANSA Developer Guide*.

[Use Special Characters to Enhance HELP Text](#) in the *LANSA for i User Guide*.

[↑ 12.6 Help Text Enhancement & Substitution Values](#)

12.6.2 Substitution/Control Values - Visual LANSA only

These help text substitutions are used to link help for other objects to the automatically generated Index (& Contents) for the current object. They do not produce visible entries in the generated help text.

Note that the Contents are only produced for the standard help interface provided with Windows.

Substitution Value	Description, Effects and Comments
---------------------------	--

<code>\$\$INDEXFLD</code> = <field-name>	Causes an item to be included in the Index that will display help text for the designated field. The rest of the current line is ignored. The label used in the Index is the field's description.
<code>\$\$INDEXCOM</code> = <object-name>, <component-name>	Causes an item to be included in the Index that will display help text for the designated component. The component is specified using the owning object (that is, a form) and component names, separated by a comma. The rest of the current line is ignored. The label used in the Index is the component's name. If no component name is specified then the index item will display help for the object (that is, the form) itself.
<code>\$\$INDEXPRO</code> = <process-name>	Causes an item to be included in the Index that will display help text for the designated process. The rest of the current line is ignored. The label used in the Index is the process's description.
<code>\$\$INDEXFUN</code> = <process-name>, <function-name>	Causes an item to be included in the Index that will display help text for the designated function. The function is specified using the owning process and function names, separated by a comma. The rest of the current line is ignored. The label used in the index is the function's description.
<code>\$\$ROOT</code>	Causes any following Index substitutions (<code>\$\$INDEX...</code>) to be added to the root of the Contents tree-view. Useful for index items that might be applicable to all or a number of topics.
<code>\$\$LEAF</code>	Causes any following Index substitutions (<code>\$\$INDEX...</code>) to be added below the current topic in the Contents tree-view. This is

the default value and only needs to be specified to switch back after \$\$ROOT has been used.

\$\$IMAGE = A bitmap (*.BMP) image file to be included in the help text at <filename>, this point. The image is centered in the screen. The default path <alternate-text> is the current partition's Execute directory, but any path may be specified. Note: Universal Naming Convention (UNC) paths are not supported. The alternate text is used instead of the image in an interface where images are not displayed, such as a 5250 terminal. If the alternate text is not specified then the filename is used instead.

Also See

[How to Use Special Characters](#) in the *Visual LANSAs Developer Guide*.

[↑ 12.6 Help Text Enhancement & Substitution Values](#)

12.6.3 Help Text Attributes

Character	Description
% (percentage)	High Intensity
{ (left parenthesis)	Underline
@ (at)	Reverse Image
~ (accent)	Blink
} (right parenthesis)	Revert to normal display
\ (backslash)	Revert to normal display

If the Help Text Attribute characters conflict with those above, you can re-assign them using these keywords:

Keyword	Example	Description
\$\$HI=	\$\$HI=!	High Intensity Causes the special character assigned for high intensity display to be re assigned to a user defined character.
\$\$RI=	\$\$RI=>	Reverse Image Causes the special character assigned for reverse image display to be re assigned to a user defined character.
\$\$BL=	\$\$BL=:	Blink Causes the special character assigned for blink display to be re assigned to a user defined character.
\$\$UL=	\$\$UL=+	Underline Causes the special character assigned for under line display to be re assigned to a user defined character.
\$\$N1=	\$\$N1=*	Normal Display Causes the special character assigned for

		normal display one to be re assigned to a user defined character.
\$\$N2=	\$\$N2=#	Normal Display Causes the special character assigned for normal display two to be re assigned to a user defined character.

Also See

[How to Use Special Characters](#) in the *Developer Guide*.

[Use Special Characters to Enhance HELP Text](#) in the *LANSA for i User Guide*.

[↑ 12.6 Help Text Enhancement & Substitution Values](#)

13. Common RDML Parameters & BIF Notes

[13.1 RDML Command Parameters](#)

[13.2 Built-In Function Notes](#)

13.1 RDML Command Parameters

13.1.1 I/O Commands

13.1.2 Field Groups and Expandable Groups

13.1.3 RDML Screen Modes and Mode Sensitive Commands

13.1.4 Specifying Conditions and Expressions

13.1.5 Arithmetic and Expression Operators

13.1.6 Quotes and Quoted Strings

13.1.7 Prompt_Key Processing

↑ 13. Common RDML Parameters & BIF Notes

13.1.1 I/O Commands

[Specifying File Names in I/O Commands](#)

[Specifying File Key Lists in I/O Commands](#)

[Specifying WHERE Parameter in I/O Commands](#)

[I/O Command Return Codes Table](#)

[I/O Status Record Locked](#)

[↑ 13.1 RDML Command Parameters](#)

Specifying File Names in I/O Commands

Most of the LANSAs database I/O commands require the specification of a file name. The parameter name may be FROM_FILE, TO_FILE, INTO_FILE, etc. but in all cases, the way that the file name is specified is identical.

The following points apply to specifying a file name in any LANSAs command:

- The file nominated must be defined within the LANSAs system as either a physical file or a logical file.
- Index-only logical files may not be used for this parameter in RDML code that will execute on the IBM i.
- Optionally a library name may be specified.

The use of the file and library name (i.e. fully qualified file names) is **NOT** recommended because it "locks" the RDML program into using a certain file in a certain library. This may cause problems when you attempt to import or export functions to/from other versions of LANSAs that use different library names.

For the IBM i, separate the file and library name using "." (i.e. full stop). For example a TO_FILE parameter might be specified as:

```
TO_FILE(CUSTMST)
TO_FILE(CUSTMST.QGPL)
TO_FILE(CUSTMST.USERLIB01)
```

For Visual LANSAs, separate the file and library name using a space. In Visual LANSAs, for example, the above TO_FILE parameters would be specified as:

```
TO_FILE(CUSTMST QGPL)
TO_FILE(CUSTMST USERLIB01)
```

Portability Considerations

On platforms other than IBM i, Visual LANSAs will ignore the library.

- If a library name is not specified a default library name called *FIRST is used. This indicates that the library list of the job in which the function is being compiled should be searched (in order) to locate the required file. If the file cannot be found using this method then the first definition of the file that can be found in the LANSAs dictionary should be used. In such cases a

warning message will be issued.

Portability Considerations Code generation varies for RDML functions and RDMLX code, and may cause a difference in which library is used where there are multiple files of the same name. RDML function generation on IBM i matches up File references to Libraries using the Library List of the job that is compiling the object. If not found in the library list, the first File in the repository found in EBCDIC collation sequence order will be used. RDMLX objects are generated on Windows, which does not have a Library List, and so the first file in the repository found in ANSI collation sequence order will be used.

[↑ 13.1.1 I/O Commands](#)

Specifying File Key Lists in I/O Commands

Many of the LANSAs database I/O commands allow the specification of a file key. In all cases the method and logic used to set up the file key is identical.

The following points should be noted about specifying file keys:

- The order that the key fields are specified on the command is as important as the content of the key fields.
- The key field nominated does not have to (and often will not) have the same name as the matching key field. The key fields nominated in the command are matched in the order specified with the actual key fields of the file.

For example, if #ORDNUM contains 123456 and #LINENO contains 1, then this command will attempt to fetch the first record in file ORDLIN with an order number = 123456 and a line number = 1:

```
FETCH FROM_FILE(ORDLIN) WITH_KEY(#ORDNUM #LINENO)
```

If, however, the command is specified as then LANSAs will attempt to fetch the first record in file ORDLIN with an order number = 1 and a line number = 123456:

```
FETCH FROM_FILE(ORDLIN) WITH_KEY(#LINENO #ORDNUM)
```

This is because the actual file keys are "order number" followed by "line number". LANSAs processes the key fields nominated, by matching their position with the actual file keys, **not** by their names.

- The key field nominated does not have to have the same length as the key field in the file. LANSAs will automatically adjust the lengths as required. However, the key field nominated and the actual file key field must be of the same type (alphanumeric or numeric).
- Most commands support the use of "partial" keys. For instance if a file is keyed by KEY01, KEY02 and KEY03 it is possible to use the following variations:

```
KEY01 KEY02 KEY03  
or KEY01 KEY02  
or KEY01
```

but, it is not possible to specify:

----- KEY03
or ----- KEY02 KEY03
or KEY01 ----- KEY03

- Expandable group expressions are allowed in key lists. The number of entries in the expanded list must match in type and must not exceed the number of fields in the key list of the file.
- When the key list contains date, time or timestamp fields, the nominated key fields must have valid date, time or timestamp values. LANSAs will validate these fields and return an error if invalid values are nominated.

Further Information

[I/O Command Return Codes Table](#)

[I/O Status Record Locked](#)

[↑ 13.1.1 I/O Commands](#)

Specifying WHERE Parameter in I/O Commands

Fields that Allow SQL Null

A field allowing SQL Null may be used as a key or as part of a where parameter just the same as any other field. It may also be compared to SQL Null. The following example shows how you might retrieve all rows in MYFILE where #MYFLD1 has a real value (not SQL Null).

```
SELECT(#MYFLDS) FROM_FILE(MYFILE) WHERE(#MYFLD1 *IsNot *Sqlnull)
```

Note that fields allowing SQL Null may behave differently in where parameters at execution time when they are SQL Null. Refer to [Assignment, Conditions, and Expressions with Fields allowing SQL Null](#) for details.

Fields of type BLOB, CLOB, Binary or VarBinary

BLOB or CLOB fields on the file cannot be used in a where condition unless being compared against *SQLNULL. For example:

```
SELECT(#MYFLDS) FROM_FILE(MYFILE) WHERE(#MYBLOB *Is *Sqlnull)
```

Any attempt to compare a BLOB or CLOB field on the file to any other value than *SQLNULL causes a FFC error.

Fields of type Float

Floats are an inaccurate numeric type. In the FFC, when a float is compared via *EQ or *NE (or equivalent) to a field, system variable (other than *ZERO), or literal value other than 0, *NULL, or *SQLNULL, a warning message is issued.

Performance

The following applies only to RDMLX on the IBM I and both RDML & RDMLX on non-IBM i platforms.

The use of the following in the WHERE condition will require it to be evaluated in the calling function or component:

- Fields not on the file
- Intrinsic functions such as IsSqlNull

For best performance, only use fields on the file in the WHERE clause. This allows the I/O module to evaluate the condition and return only the rows

matching the WHERE, rather than having to return all the rows matching the key provided to the calling function or component.

For even better performance, only use real fields on the file (not virtual fields). If the I/O command is handled in SQL, this will minimise the number of rows to be returned.

[↑ 13.1.1 I/O Commands](#)

I/O Command Return Codes Table

Command	I/O Error	Dictionary Validation	Not Found	Found Or Completed
INSERT	ER	VE #1	-	OK
UPDATE	ER #3	VE	NR	OK
FETCH	ER #3		NR	OK
SELECT	ER #3		EF #2	OK
FILECHECK -			NE	EQ
CHECK_FOR -			NE	EQ
DELETE	ER #3	VE	NR	OK

#1 An attempted INSERT of a duplicate key will return VE.

#2 A SELECT command using a WHERE parameter will select each record and test for the condition. When the last record is selected the processing will leave the SELECT loop with the data from the last record selected. This record may not have met the WHERE condition.

#3 [I/O Status Record Locked](#)

[↑ 13.1.1 I/O Commands](#)

I/O Status Record Locked

In addition to the return codes in the [I/O Command Return Codes Table](#), the system variable *DBMS_RECORD_LOCKED can be used to distinguish between an I/O error status of record locked and other I/O errors.

The following example shows how the IO_ERROR parameter passes control to the label TST when an error occurs and the GOTO NXT by-passes the if condition when the command was successful.

```
UPDATE  FIELDS(#ORDERQTY) IN_FILE(ORDLINE)
        WITH_KEY(#ORDER #LINE) IO_ERROR(TST)
GOTO    LABEL(NXT)
TST IF   COND('*DBMS_RECORD_LOCKED *EQ Y')
        MESSAGE MSGTXT('Order line record locked')
        ..... ..Required action
        ELSE
        ABORT   MSGTXT('Fatal I/O error on ORDERLINE file')
        ENDIF
NXT     ..... ..Next action
```

Comments / Warnings

If using this method on files which were compiled prior to Release 7.0, the I/O module must be recompiled first.

As the IO_ERROR parameter passes control to the label nominated, the condition should always have an ELSE command with an appropriate action to handle a non-record locked I/O error as in the example:

DBMS_RECORD_LOCKED only checks the status of the file, which is the subject of the command. If in the example batch control logic is used and the batch control record was locked *DBMS_RECORD_LOCKED would return a value of 'N'. The same applies to any files used by triggers. If the record locked status is to be checked for a file used by a trigger the above logic should be inserted into the trigger function.

Portability **IBM i**
Considerations : Automatically unlocks a file after a certain period of time.
Non-
IBM i

: This feature is emulated and disabled by default. For further details, refer to [Lock Timeout](#).

[↑ 13.1.1 I/O Commands](#)

13.1.2 Field Groups and Expandable Groups

The GROUP_BY command is used to group one or more fields under a common name. It is one of the most time saving of all the RDML commands because it saves having to repeatedly specify a long list of field names.

Most commands that require a list of field names as a parameter also allow a group name to be specified. Consider the following example:

```
BEGIN_LOOP
REQUEST  FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)
INSERT   FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)
         TO_FILE(A)
INSERT   FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)
         TO_FILE(B)
UPRINT   FIELDS(#ORDLIN #PRODUCT #QUANTITY #PRICE)
CHANGE   FIELD(#ORDLIN #PRODUCT #QUANTITY #PRICE)
         TO(*DEFAULT)
END_LOOP
```

Now consider the identical RDML program written using a GROUP_BY command to group all the fields under a common name:

```
GROUP_BY NAMED(#ORDERLINE) FIELDS(#ORDLIN #PRODUCT #Q
                                     #PRICE)

BEGIN_LOOP
REQUEST  FIELDS(#ORDERLINE)
INSERT   FIELDS(#ORDERLINE) TO_FILE(A)
INSERT   FIELDS(#ORDERLINE) TO_FILE(B)
UPRINT   FIELDS(#ORDERLINE)
CHANGE   FIELD(#ORDERLINE) TO(*DEFAULT)
END_LOOP
```

If 5 new fields were to be added to the RDML program which would be the easiest to change?

Some points to note about groups and the GROUP_BY command are:

- As many GROUP_BY commands as desired, can be declared.
- A field can be declared in the FIELDS parameter of multiple GROUP_BY commands.

- Generally a GROUP_BY name can be used wherever a list of field names can be specified.
- If a GROUP_BY name is declared like this:

```
GROUP_BY NAMED(#ORDERHEAD) FIELDS(#ORDER #CUSTNO #AI
        #ADDR2
        #POSTCD)
```

and then used like this:

```
FETCH FIELDS(#ORDERHEAD) FROM_FILE(ORDHED)
FETCH FIELDS(#ORDERHEAD) FROM_FILE(CUSMST)
```

then the first FETCH will only retrieve fields in the group that come from the ORDHED file. In this case only #ORDER, and #CUSTNO will be fetched. All other fields in the group are unchanged / ignored by the command because they don't come from the ORDHED file.

The second FETCH will retrieve the fields #CUSTNO (again), #ADDR1, #ADDR2 and #POSTCD because they all come from the CUSMST file. Field #ORDER will be unchanged / ignored by this command as it does not come from the CUSMST file.

Also See

[Expandable Groups](#)

[Special Considerations for Expandable Groups](#)

[Expandable Group Expressions](#)

[Expandable Group Examples](#)

[↑ 13.1 RDML Command Parameters](#)

Expandable Groups

Expandable groups are a special kind of group whose most important feature is that they themselves can be assembled of other expandable groups. Expandable groups names must start with the prefix '#XG_' (For example, #XG_GROUP1).

In addition, expandable groups also support the special values *INCLUDING and *EXCLUDING which allows the exclusion of specific fields from a field list.

They can be used in most places where a list of fields or a group name are allowed. Because an expandable group is replaced by the expanded field list it represents (That's why they are called expandable groups), it can be used in field lists which do not allow conventional groups. Refer to the help text of the RDML command's parameter to find out if it allows expandable group name(s).

[↑ 13.1.2 Field Groups and Expandable Groups](#)

Special Considerations for Expandable Groups

- Fields starting with the same prefix as expandable groups (#XG_) cannot be included in expandable groups. When an entry starting with this prefix is found in the FIELDS parameter of an expandable group GROUP_BY definition, the RDML full function checker will handle it as a group name, not as a field name.
- Expandable groups must be defined before they can be referenced. In this aspect, expandable groups differ from conventional groups in that the placement of the GROUP_BY definition within the function is important.
- Conventional groups are not allowed in the FIELDS parameter of an expandable group.
- Attributes assigned to fields in expandable groups are ignored. Refer to [RDML Field Attributes and their Use](#) for details.
- The following limits apply:
 - Up to 100 expandable groups can be defined in a function.
 - Up to 100 fields can be grouped into an expandable group. At no time during the expansion can the field list go beyond this limit, even if some fields will be excluded from the list later in the expansion.
 - Up to 999 fields altogether can be defined in expandable groups within a function.

Expandable groups may be defined as follows:

```
GROUP_BY NAMED(#XG_CUST) FIELDS(#CUSTNO #CUSTNM #ADD
#POSTCD)
```

```
GROUP_BY NAMED(#XG_PRKY) FIELDS(#GROUP #PRODNO)
```

```
GROUP_BY NAMED(#XG_PROD) FIELDS(#XG_PRKY #DESC #CLASS
#BARCOD)
```

```
GROUP_BY NAMED(#XG_ORD) FIELDS(#XG_CUST #XG_PROD)
```

and then used like this:

```
FETCH FIELDS(#XG_CUST) FROM_FILE(CUSMST)
```

```
FETCH FIELDS(#XG_PROD) FROM_FILE(PRDMST) WITH_KEY(#XC
```

```
DISPLAY FIELDS(#XG_ORD)
```

[↑ 13.1.2 Field Groups and Expandable Groups](#)

Expandable Group Expressions

An expandable group expression is a list of entries which can contain any of:

- Field name
- Expandable group name
- Alphanumeric or numeric literal
- System variable
- Expandable group special value, which can be any of the following:
 - ALL, specifies that all fields from the currently active version of the file in context be expanded in the field list.
 - ALL_REAL, specifies that all real fields from the currently active version of the file in context be expanded in the field list.
 - ALL_VIRT, specifies that all virtual fields from the currently active version of the file in context be expanded in the field list.
 - EXCLUDING, specifies that fields following this special value must be excluded from the field list.
 - INCLUDING, specifies that fields following this special value must be included in the field list. This special value is only required after an *EXCLUDING entry has caused the field list to be in exclusion mode.
- The special values *ALL, *ALL_REAL and *ALL_VIRT are only valid in the context of a file (e.g. when fetching a record using the FETCH command.)
- The usual parameter restrictions also apply. For example, the FIELDS parameter of the FETCH command doesn't allow constants or system variables.

[↑ 13.1.2 Field Groups and Expandable Groups](#)

Expandable Group Examples

Example 1:

```
GROUP_BY (#XG_ADDR) FIELDS (#ADDR1 #ADDR2 #ADDR3)
FETCH (*ALL *EXCLUDING #XG_ADDR *INCLUDING #ADDR1)
FROM_FILE (CUSTMST)
```

Example 2:

```
GROUP_BY (#XG_GROUP1) FIELDS (#A #B #C)
GROUP_BY (#XG_GROUP2) FIELDS (#D #E #F)
SUBMIT PROCESS (PROC01) FUNCTION (FUN01)
EXCHANGE (#XG_GROUP1 *SYSUAR1 #XG_GROUP2 *EXCLUD
```

[↑ 13.1.2 Field Groups and Expandable Groups](#)

13.1.3 RDML Screen Modes and Mode Sensitive Commands

The DISPLAY and POP_UP commands cause information to be displayed on a workstation. In addition the "mode" in which the information should be displayed can also be specified. Currently screens can be displayed in 4 different "modes" that are referred to as:

- *DISPLAY mode
- *CHANGE mode
- *ADD mode
- *DELETE mode

The processing mode which is in use when a DISPLAY and POP_UP command is executed affects:

- Whether or not fields on the screen can be changed by the user.
- The function keys that are enabled.
- The processing logic that is used by the DISPLAY and POP_UP commands.

The screen mode can be set / changed by using the SET_MODE command and can be tested using the IF_MODE command.

Also See

[Modes and Fields that Can Be Changed](#)

[Modes, Function Keys and Processing Logic](#)

[Mode Sensitive Commands](#)

[↑ 13.1 RDML Command Parameters](#)

Modes and Fields that Can Be Changed

The following table indicates how the screen processing mode affects whether or not fields on the screen can be changed.

Fields that use the *INOUTCOND field conditioning attribute to control when a field can or cannot be changed are **not** subject to the rules specified in this table.

Mode	Fields That Can Be Changed	Fields That Cannot Be Changed
	"	"
	INPUT CAPABLE	"
	"	PROTECTED FIELDS
		"
*DISPLAY	Those with *INPUT attribute	All others
*CHANGE	All others	Those with *NOCHG attribute
*ADD	All others	Those with *NOCHG attribute
*DELETE	Those with *INPUT attribute	All others

Note that when a list is initialized (with the INZ_LIST command), or has an entry added (with the ADD_ENTRY command) or updated (with the UPD_ENTRY command) the mode is "set" when the command is executed, NOT when the list is displayed.

For instance the commands:

```
SET_MODE TO(*DISPLAY)
ADD_ENTRY TO_LIST(#BROWSE)
```

or just

```
ADD_ENTRY TO_LIST(#BROWSE) WITH_MODE(*DISPLAY)
```

and then

```
SET_MODE TO(*CHANGE)
ADD_ENTRY TO_LIST(#BROWSE)
```

or just

```
ADD_ENTRY TO_LIST(#BROWSE) WITH_MODE(*CHANGE)
```

would cause the first entry in the list to be "protected" and the second entry in the list to be "input capable". This is because the mode of a list entry is "set" at the time the entry is added to (or updated into) the list, not at the time that the list is displayed.

The SELECTLIST command has options that allow only entries in a list to be processed according to the "mode" that was active when the entry was added or updated last. Refer to the SELECTLIST command for more information.

Note that all list processing commands (e.g.: ADD_ENTRY or UPD_ENTRY) have an optional parameter to allow the mode of the individual list entry to be set, regardless of the current mode of the program. This is illustrated by the "or just" examples above.

[↑ 13.1.3 RDML Screen Modes and Mode Sensitive Commands](#)

Modes, Function Keys and Processing Logic

The following table indicates which function keys are enabled when a particular processing mode is in use and what happens when the function key is used.

Note that to be "**enabled**" a function key must match the entry in the following table and be specified with the *YES option on the associated command. Refer to the relevant EXIT_KEY, MENU_KEY, ADD_KEY, CHANGE_KEY, DELETE_KEY and PROMPT_KEY parameters for more information.

Additionally, the function key must not have been "**disabled**" by the specification of a controlling "condition" (previously defined by a DEF_COND command) that is not true at the time the screen panel is processed.

Finally, if a controlling condition has been specified to "enable" or "disable" the function key, it **cannot override** the LANSAs mode sensitive **disabling** of the key. For instance the CHANGE key will not be enabled in *ADD mode, even if the controlling condition is true.

Mode	Function Key	Enabled	What Happens When Used
*DISPLAY	EXIT/SYSTEM	YES	Function ends or control passed to nominated command. Mode remains unchanged.
	MENU/CANCEL	YES	Process menu re-displayed or control passed to nominated command. Mode remains unchanged.
	MESSAGES	YES	Messages displayed, then current screen re-displayed. Mode remains unchanged.
	PROMPT	YES	Prompt request processed, then current screen re-displayed. Mode remains unchanged.
	ADD	YES	Mode changed to *ADD. Control is passed to nominated label.
	CHANGE	YES	Mode is changed to *CHANGE. Current screen is re-displayed with input capable fields to allow changes to

			be made.
	DELETE	YES	Mode is changed to *DELETE. Current screen is re-displayed with a message requesting that the delete request be confirmed.
*CHANGE	EXIT/SYSTEM	YES	Function ends or control passed to nominated command. Mode remains unchanged.
	MENU/CANCEL	YES	Process menu re-displayed or control passed to nominated command. Mode remains unchanged.
	MESSAGES	YES	Messages displayed, then current screen re-displayed. Mode remains unchanged.
	PROMPT	YES	Prompt request processed, then current screen re-displayed. Mode remains unchanged.
	ADD	NO	Not enabled when screen is in *CHANGE mode.
	CHANGE	NO	Enabled only when screen is in *CHANGE mode.
	DELETE	NO	Not enabled when screen is in *CHANGE mode.
*ADD	EXIT/SYSTEM	YES	Function ends or control is passed to nominated command. Mode remains unchanged.
	MENU/CANCEL	YES	Process menu re-displayed or control passed to nominated command. Mode remains unchanged.
	MESSAGES	YES	Messages displayed, then current screen re-displayed. Mode remains unchanged.

PROMPT	YES	Prompt request processed, then current screen re-displayed. Mode remains unchanged.
ADD	NO	Enabled only when screen is in *ADD mode.
CHANGE	NO	These keys are not enabled when screen is in *ADD mode.
DELETE	NO	
*DELETE EXIT	YES	Function ends or control passed to nominated command. Mode remains unchanged.
MENU	YES	Process menu re-displayed or control passed to nominated command. Mode remains unchanged.
MESSAGES	YES	Messages displayed, then current screen re-displayed. Mode remains unchanged.
PROMPT	YES	Prompt request processed, then current screen re-displayed. Mode remains unchanged.
ADD	NO	These keys are not enabled when screen is in *DELETE mode.
CHANGE	NO	
DELETE	NO	

↑ 13.1.3 RDML Screen Modes and Mode Sensitive Commands

Mode Sensitive Commands

The following commands are "mode sensitive":

DISPLAY

POP_UP

ADD_ENTRY

UPD_ENTRY

INZ_LIST

The following command is **not** "mode sensitive":

REQUEST

[↑ 13.1.3 RDML Screen Modes and Mode Sensitive Commands](#)

13.1.4 Specifying Conditions and Expressions

Many of the RDML commands in the LANSA system require that a condition or an expression be specified.

Generally a **condition** is a statement that can be evaluated to produce a "true" or "false" answer.

Refer to the online *Technical Reference* for details of this topic and of *Arithmetic and Expression Operators*.

For instance "#A *LT 10" is a condition. Either field #A is less than 10 (true) or it isn't (false). A condition may contain the following operators for RDML commands:

*GT,>

*LT,<

*EQ,=

*NE,^=

*GE,>=

*LE,<=

We recommend the use of the first operator for cross-platform use. That is, "#A *LT 10", not "#A < 10"

An **expression** is a statement that can be evaluated to produce either a numeric or an alphanumeric result. For instance the expression "(#X + 10) / 2" produces a numeric result which is the sum of #X and 10 divided by 2. Note that in this case the result has no "true" or "false" meaning. The result is just a number.

Often an expression is contained within a condition. Consider the condition:

```
#A < ((#B + 10.62) / 3.14)
|   |               ||
|   |               ||
|   --- expression --- |
|                       |
|----- condition-----
```

The components of a condition or an expression can be:

- An alphanumeric literal such as 'NSW', NSW, 'Balmain' or BALMAIN.

- A numeric literal such as 1, 14.23, -1.141217.
- Another field name such as #CUSTNO, #INVNUM, etc.
- A system variable name such as *BLANKS, *ZERO, *DATE or any other system variable defined at your installation.
- A process parameter such as *UP01, *UP02, etc.

A full RDMLX object allows RDMLX [Enhanced Expressions](#). Enhanced expressions add support for:

- Methods, intrinsic functions, and properties
- Additional operators *Not, *IS, *ISNOT, *IsEqual, *IsOfType, *AS, *ANDIF and *ORIF

Note that alphanumeric literals do NOT have to be in quotes when used in a condition or an expression. Quotes are only required when the alphanumeric literal contains lowercase characters. If no quotes are used the alpha literal is converted to uppercase. Thus BALMAIN = balmain = Balmain = balMAIN, however, Balmain does not equal 'Balmain'.

Note also that field names must be preceded by a # (hash) symbol when used in conditions or expressions. This allows LANSAP to differentiate between fields and alphanumeric literals. For instance the expression CNTRY = AUST does not indicate which of the components is the field and which is the alphanumeric literal. The correct format is #CNTRY = AUST.

Note also that fields allowing SQL Null may behave differently in conditions and expressions at execution time when they are SQL Null. Refer to [Assignment, Conditions, and Expressions with Fields allowing SQL Null](#) for details.

[↑ 13.1 RDML Command Parameters](#)

13.1.5 Arithmetic and Expression Operators

Within an expression or condition a set of operators can be used. These are as follows:

Operator	Description
(Open bracket
)	Close bracket
+	Add
-	Subtract
/	Divide
*	Multiply
=	Compare equal
^=	Compare not equal. See Note :
<	Compare less than
<=	Compare less than or equal to
>	Compare greater than
>=	Compare greater than or equal to
*EQ	Compare equal
*NE	Compare not equal
*LT	Compare less than
*LE	Compare less than or equal to
*GT	Compare greater than
*GE	Compare greater than or equal to
AND	And
OR	Or
*AND	And

*OR Or

Expression evaluation is **left to right within brackets**, so use brackets **liberally** whenever any doubt exists as to the order in which the expression will be evaluated.

The liberal use of brackets is a **good programming practice** as well. It makes clear your intent to the RDML compiler, but also more importantly, to anyone maintaining the application in the future.

Expression components are checked for type and length compatibility. The syntax of the expression or condition is also checked to ensure that it is correct.

Since all conditions and expressions specified under LANSAs are "quoted strings" you should also read [13.1.6 Quotes and Quoted Strings](#).

Some examples of conditions and expressions are:

- Condition RDML commands to execute only if field #A is less than 10:

```
IF COND('#A < 10')  
or IF COND('#A *LT 10')
```

- Change field #A to contain the value 10:

```
CHANGE FIELD(#A) TO(10)  
or CHANGE #A (10)  
or CHANGE #A 10
```

- Condition RDML commands to execute only if field #A is greater than the sum of field #B and 10.62 divided by 2:

```
IF COND('#A < ((#B + 10.62) / 2)')  
or IF ('#A < ((#B + 10.62) / 2)')  
or IF '#A < ((#B + 10.62) / 2)'
```

- Change field #A to contain the sum of field #B and 10.62 divided by 2:

```
CHANGE FIELD(#A) TO('(#B + 10.62) / 2')  
or CHANGE #A ('(#B + 10.62) / 2')  
or CHANGE #A '#B + 10.62) / 2'
```

- Request that a product number be input by the user until it can be found in the product master, then display full details of the product:

```
GROUP_BY NAME(#PRODUCT) FIELDS(#PRODNO #DESC #PRIC
      #QOH #TAX)
BEGIN_LOOP
DUNTIL COND('#IO$STS = OK')
REQUEST FIELD(#PRODNO)
FETCH FIELDS(#PRODUCT) FROM_FILE(PROMST)
      WITH_KEY(#PRODNO)
ENDUNTIL

DISPLAY FIELDS(#PRODUCT)
END_LOOP
```

Note:

Due to translation table issues between IBM i and PC platforms (ASCII/EBCDIC), using 5250 terminals or 5250 emulation mode terminals users should be very careful when using the ^= Compare not equal expression, which can be presented as ^= or ¢= or ¬= depending on the terminal/keyboard used during edit. Use the *NE expression instead.

[↑ 13.1 RDML Command Parameters](#)

13.1.6 Quotes and Quoted Strings

Some RDML commands require that associated parameters appear as "quoted strings" because LANSAs uses the IBM i operating system command definition and prompting facilities.

Command parameter with imbedded blanks

For example, to increment field #COUNT by 1 the correct format is:

```
CHANGE FIELD(#COUNT) TO('#COUNT + 1')  
or CHANGE #COUNT '#COUNT + 1'  
or CHANGE #COUNT ('#COUNT + 1')  
or CHANGE #COUNT '('#COUNT + 1')
```

but NOT CHANGE #COUNT (#COUNT + 1)

This is because the IBM i command facilities demand that a command parameter be enclosed in quotes if it contains imbedded blanks. In this case the string "#COUNT + 1" definitely contains imbedded blanks and thus **must** be enclosed in quotes (ie: made into a "quoted" string).

Quotes within a quotes string

When LANSAs processes the command only the part between the quotes (but not the quotes themselves) are passed to LANSAs by the operating system.

The matter is complicated even further if you wish to use quotes within a "quoted" string. This situation usually arises when coding IF or CASE conditions.

The rule for using quotes inside a quoted string is: use 2 quotes instead of just one.

For instance, to check if #FIELD contains a lowercase "a" you would have to code:

```
IF '#FIELD = "a"'
```

What is passed to LANSAs by the IBM i command processor as the expression associated with the IF command is actually:

```
#FIELD = 'a'
```

because the operating system does not pass the outer quotes and replaces

occurrences of 2 quotes within the string with just one quote.

Simple guidelines for quotes

However, the handling of quotes within LANSAs can be made much easier by following 2 simple guidelines:

1. Only use quotes inside a quoted string when absolutely necessary.
2. Use the formatted prompting facilities to input complex quoted strings.

With regard to **point 1**, LANSAs does not require that alphanumeric literals be quoted. Thus the following are **identical** conditions because alphanumeric literals that are not enclosed in quotes are converted to uppercase:

```
IF '#FIELD = A'  
IF '#FIELD = a'  
IF '#FIELD = "A"
```

Only use quotes around alphanumeric literals if the test involves lowercase characters. For instance to test for a lowercase "a" in #FIELD:

```
IF '#FIELD = "a"
```

With further regard to **point 2**, you will find that the formatted prompting facility will automatically insert the required outer quotes. For instance if you prompt an IF command and enter the condition as:

```
#FIELD = A
```

then the prompter will automatically re-format the condition so that it is a valid "quoted" string.

The final version of the command created by the prompter would look like this:

```
IF COND('#FIELD = A')
```

The same applies when it is necessary to use quotes within the expression. For instance if you specify to the prompter the following condition:

```
#FIELD = 'a'
```

then it will re-format the command automatically and insert the necessary inner and outer quote symbols. The command created by the prompter would look like this:

```
IF COND('#FIELD = "a"')
```

Quoted strings as parameter values

On the RDML CALL command, care should be taken if using quoted strings as parameter values. A quoted string has to be enclosed in triple quotes. For example, to use 'ABC' as the parameter value, enter ""ABC"".

When the function containing the CALL is compiled the RPG generator uses the length of the string, including the single quotes, to create the parameter fields. For example if program A has two alpha parameters both 5 characters long and a call is inserted in function B as:

```
CALL PGM(A) PARM('ABC' ""ABC"")
```

The generator will create a 3 character parameter (ABC) and a 5 character parameter ('ABC') for the call. When function B is executed program A may end in error because the 4th and 5th characters of the first parameter within program A will contain garbage.

Triple quotes with first character that looks like a number

For information, please refer to the [Change command's Comments and Warnings](#).

[↑ 13.1 RDML Command Parameters](#)

13.1.7 Prompt_Key Processing

The commands DISPLAY, REQUEST and POP_UP all have a parameter called PROMPT_KEY. The default value for this parameter looks like this:

```
PROMPT_KEY(*DFT *AUTO)
```

The **first value** in the parameter indicates whether or not the prompt key should be enabled. Allowable values are *YES, *NO and *DFT. The special value *DFT indicates that it should be enabled or disabled according to byte 477 of the system definition data area DC@A01. For full details of the layout of the [System Definition Data Areas](#) refer to the *LANSA for i User Guide*.

The **second value** indicates what should happen when the prompt key is used. Allowable values are *NEXT (control should pass to the next RDML command), a command label (indicating that control should be passed to the label) or *AUTO (which indicates that the function key should be handled automatically by LANSA).

In the first two cases (***NEXT or a label**) the prompt key handling is controlled entirely by the RDML program, and thus what happens is entirely at the discretion of the programmer.

It is the final case (***AUTO**) that is of interest here. Most of the following material discusses how the prompt key is handled automatically.

Finally, the actual function key number assigned to the prompt key is set like this:

- In **non-SAA/CUA applications** it is assigned from bytes 478 to 479 of the system definition data area. Refer to the *LANSA for i User Guide* for full details of the [System Definition Data Areas](#).
- In **SAA/CUA applications** it is assigned from the partition level value assigned to the prompt function key. Refer to [Steps to Create or Change a Partition](#) in the *LANSA for i User Guide* for more details of how and when this value is assigned.

↑ [13.1 RDML Command Parameters](#)

13.2 Built-In Function Notes

[13.2.1 Database Connection](#)

[13.2.2 Email Built-In Functions Notes](#)

[13.2.3 Zip Built-in Functions](#)

[↑ 13. Common RDML Parameters & BIF Notes](#)

13.2.1 Database Connection

DEFINE_DB_SERVER can be used in isolation from the other BIFs to just override the connection parameters and database type or it can be used with the full set of related BIFs in this sequence:

DEFINE_DB_SERVER, CONNECT_SERVER, CONNECT_FILE.

The full set of BIFs is only needed when the OTHER File is in a database with a different DSN to the one with which it was loaded.

To connect to all the default databases on startup, that is all the databases defined in the OAMs, it is necessary to execute DEFINE_DB_SERVER (with database type specified) and CONNECT_SERVER for each database using the DSN that is in each OAM. CONNECT_FILE is not required if the DSN is the same as in the OAM. That is, an OTHER File is implicitly connected to the database from which it was loaded.

If your environment has a development, test and production version strategy, the simplest way of managing the differing locations of Other Files as the application passes through the various stages, is to use the same ODBC DSN but alter the definition of it to point to a different physical database. Thus, the default database embedded in the OAM will access a different physical database.

It may seem a simple thing to switch databases at will, but its not if any OAMs are shared. That is, if you are using the same file in multiple databases. When switching a database and there are shared OAMs you must close every file that has been used, including Code-File Lookups, triggers, etc. Its easy to miss a file. If you don't do this the original database will be accessed. That is, the database is set when the File is opened. After that, all IO will go to the original database. When using a single form, a CLOSE is all that is required to close all the files that have been used. When multiple objects are used, its far more complex. Its essential that all Components and Functions exit back to the initial Component/Function, ensuring that any HEAVYUSAGE objects call CLOSE before returning. In fact using LIGHTUSAGE Functions and Dynamic Components everywhere may be the best solution. The following is an example of the code that would need to be executed in the initial component if that component accesses any files:

```
Subroutine Name(SwitchSRV) Parms(#SwitchSRV)
Define Field(#LastSRV) Reffld(#SERVER1)
```

Close

```
If (#Switchsrv *NE #Lastsrv)
If (#LastSRV *NE *BLANK')
Use Builtin(Disconnect_file) With_Args(* #LastSRV)
Endif
Use Builtin(connect_file) With_Args(* #SwitchSRV)
Endif

#LastSRV := #SwitchSRV
Endroutine
```

Note that if you have multiple databases connected with files under commitment control, a COMMIT or ROLLBACK will commit or rollback all transactions on all databases.

[↑ 13.2 Built-In Function Notes](#)

13.2.2 Email Built-In Functions Notes

MAIL Built-In Functions

DO NOT support Microsoft Office 64 bit.

LANSA is a 32 bit application and so it cannot interact with the ActiveX/MAPI interfaces of 64 bit office.

DO support the 32 bit installation of Office on a 64 bit machine however, and this is the default installation option.

An email message that is to be sent must be constructed commencing with a call to MAIL_START followed by other calls in the Email Handling series to add the data. The message is then sent using the MAIL_SEND call.

Notes:

- Only use Email Handling Built-In Functions in applications that are to fully execute under the control of IBM i or Windows.
- In the Windows environment, these Built-In Functions should not be used in functions that are invoked on remote Windows server systems.
- For Email BIFs to work locally, you must set the PROFILE correctly. To find the correct PROFILE, go to *Outlook*, select *Tools* then *Options* and click on the *Mail Services* tab.

[↑ 13.2 Built-In Function Notes](#)

13.2.3 Zip Built-in Functions

The ZIP BIFS utilize Info-ZIP's compression and decompression utilities via unzip32.dll, zip32.dll and unzipsfx.exe. Info-ZIP's software (Zip, UnZip and related utilities) is free and can be obtained as source code or executables from [info-zip's web site](#). Also refer to Info-zip's web site for information on the .ZIP file format and functionality of zip and unzip.

PKWARE introduced the .ZIP file format in 1989. According to strict interpretation of the zipfile specification (as specified by PKWARE and amended by Info-ZIP), the following limits apply to all zipfile archives:

Limit	Maximum
Number of Files	65536
Uncompressed size of a single file	4 GB
Compressed size of a single file	4 GB
Maximum size of archive that can be created	256 TB
Maximum size of archive that can be extracted	4 GB

Using the wildcard * in file specifications: If you want to include (or ignore) all files that are named readme or readme.* then add readme* to the list (rather than readme.*).

[↑ 13.2 Built-In Function Notes](#)

14. Template Commands and Variables

See also:

[14.22 Tips for Template Programming](#)

- 14.1 [@@CLR_LST Command](#) Clears a working list
- 14.2 [@@CMP_IDX Command](#) Compares an index value
- 14.3 [@@COMMENT Command](#) Adds a comment line to the generated RDML code
- 14.4 [@@DEC_IDX Command](#) Decreases an index value
- 14.5 [@@GET_FILS Command](#) Gets the files to be used in this template
- 14.6 [@@GOTO Command](#) Passes control to a label
- 14.7 [@@IF Command](#) Tests a condition and pass control to a label
- 14.8 [@@INC_IDX Command](#) Increases an index value
- 14.9 [@@LABEL Command](#) A label which is the subject of another command
- 14.10 [@@MAK_LSTS Command](#) Makes a list of selected fields from another list(s)
- 14.11 [@@MRG_LSTS Command](#) Merges a list(s) with another list(s)
- 14.12 [@@QUESTION Command](#) Asks a question and receives a reply
- 14.13 [@@RTV_FLDS Command](#) Retrieves the fields in a file
- 14.14 [@@RTV_KEYS Command](#) Retrieves the keys of a file
- 14.15 [@@RTV_RELN Command](#) Retrieves the relationship between two files
- 14.16 [@@SET_IDX Command](#) Sets an index value

[14.17 General Template Variables](#)

[14.18 Question and Answer Template Variables](#)

[14.19 File Template Variables](#)

[14.20 List Template Variables](#)

[14.21 Template Error Messages](#)

[Application Template Program Examples](#) in the *Visual LANSAs Developer Guide*.

14.1 @@CLR_LST Command

The @@CLR_LST command is used to create/clear a specified list.

NOTE: Before any command involving a list is used, the list must have already been created by an @@CLR_LST. It is suggested that all lists are cleared at the end of all application templates so that all work records will be deleted from the work file on completion of the application template.

Required

```
@@CLR_LST ----- NUMBER ---- number -----  
--|  
      index
```

Parameters

NUMBER

Specifies the list number to be created and/or cleared. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes.

Examples

The following examples apply to the @@CLR_LST command.

Example 1: Clear list number 1:

```
@@CLR_LST NUMBER(1)
```

Example 2: Clear list indexed by index name "CF":

```
@@CLR_LST NUMBER(CF)
```

14.2 @@CMP_IDX Command

The @@CMP_IDX command is used to compare an index value with a numeric value. This index name may be a new or existing index. If the index is a new index name, then the index value will be set to 1. The maximum number of indexes which can be used in an application template is 999.

Required

```
@@CMP_IDX ----- IDX_NAME ----- name -----  
--->
```

```
>--- IDX_VALUE ---- value ----->  
      numeric variable
```

Optional

```
>-----IF_LT ----- label ----->
```

```
>-----IF_GT ----- label ----->
```

```
>-----IF_EQ ----- label -----|
```

Parameters

IDX_NAME

Specifies the new or existing index name. The first character of the two character index name must be non-numeric.

IDX_VALUE

Specifies the numeric value to be compared with the index value. This may be any valid numeric variable, any valid index name or any valid numeric literal.

IF_LT

Specifies the label of the command which is to receive control if the value of the index is less than the comparison value. The label specified in this parameter must be the label of one and only one other command in the application template.

IF_GT

Specifies the label of the command which is to receive control if the value of the index is greater than the comparison value. The label specified in this parameter must be the label of one and only one other command in the application template.

IF_EQ

Specifies the label of the command which is to receive control if the value of the index is equal to the comparison value. The label specified in this parameter must be the label of one and only one other command in the application template.

14.2.1 @@CMP_IDX Examples

The following examples apply to the @@CMP_IDX command.

Example 1: Compare index CF with the maximum file chosen. If greater than the maximum file number then transfer control to label LB1.

```
@@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) IF_GT(LB1)
```

Example 2: Compare index AB with the literal 10. If equal to 10 then transfer control to label A25.

```
@@CMP_IDX IDX_NAME(AB) IDX_VALUE(10) IF_EQ(A25)
```

Example 3: Compare index NE with the number of elements in list number 01. If greater than this value then transfer control to label X10.

```
@@CMP_IDX IDX_NAME(NE) IDX_VALUE(@@LNE01) IF_GT(X10)
```

14.3 @@COMMENT Command

The @@COMMENT command is used to add a comment line to the RDML code generated by this application template.

The text for a comment may contain special variables for substitution.

Comments which apply to the application template itself are entered using SEU in the standard manner for comments in CL programs, i.e.

```
/* .....comment.line.....*/
```

It is very important for application templates to be documented in this manner.

Required

```
@@COMMENT -----COMMENT ----- 'text' -----  
---|  
      |      |  
      - 10 max -
```

Parameters

COMMENT

Specifies the text for the comment to be added to the RDML code. This text may contain special variables (refer to the [14.22.5 Special Template Variable Notes](#) on special variables for more details). The comment may be up to a maximum of 10 lines of 55 characters.

Examples

The following examples apply to the @@COMMENT command.

Example 1: Add a comment line containing the current date and time to the RDML code generated.

```
@@COMMENT COMMENT('Date - @@DATE   Time - @@TIME')
```

Example 2: Add a comment line containing the name of the file used in a FETCH command.

```
@@COMMENT COMMENT('Fetch file @@FNAME01 details  ')  
FETCH   FIELDS((#HEADER)) +
```

FROM_FILE(@@FNAME01) +
WITH_KEY(@@LST03) NOT_FOUND(R10) +
ISSUE_MSG(*YES)

14.4 @@DEC_IDX Command

The @@DEC_IDX command is used to decrement an index by 1. This index name may be a new or existing index. If the index is a new index name, then the index value will be set to -1. The maximum number of indexes which can be used in an application template is 999.

Required

```
@@DEC_IDX ----- IDX_NAME ----- name -----  
--|
```

Parameters

IDX_NAME

Specifies the new or existing index name. The first character of the two character index name must be non-numeric.

Examples

The following example applies to the @@DEC_IDX command.

Example 1: Decrement index CF.

```
@@DEC_IDX IDX_NAME(CF)
```

14.5 @@GET_FILS Command

The @@GET_FILS command is used to ask for file(s) to be selected for use in the application template.

If more than 1 file can be selected, then after choosing the primary file, the user will be presented with a second list of all those files related in some manner to this selected primary file. This list will be in the form of a "tree" of relationships. For this "tree" of relationships to be built, all files involved will have to have been set up with access routes showing how all the files are related. If you have used data modeling to design your database then this will already have been done for you, otherwise you will have to manually add all access routes.

Some of the rules in selecting from this list are:

- When SGL_ONLY(*NO) has been specified, only one 1 : N (many) relationship may be chosen from the list displayed. Multiple 1 : 1 relationships may be selected.
- All higher levels in a relationship chain must be chosen before a lower level relationship can be selected.
- A maximum of (TO - FROM + 1) files may be selected.

This command creates/updates special variables @@TFMX and @@TFMN to contain the maximum and minimum file numbers chosen.

Optional

```
@@GET_FILS ----- FROM ----- 1 ----->
>
      (number 1 - 50)

>---- TO ----- 50 ----->
      (number 1 - 50)

>---- PHY_ONLY---- *YES ----->
      *NO

>---- SGL_ONLY---- *YES ----->
      *NO

>---- PROMPT ----- 'text' ----->
```

```
>---- EXTEND ---- 'text' ----->

-- 8 max --

>----HELPIDS ---- HELP panel identifiers -----|
|                                     |
----- 10 max -----
```

Parameters

FROM

Specifies the first file number to allocate to a user selected file. The default value is 1 and the maximum value is 50.

TO

Specifies the last file number to allocate to a user selected file. The default value is 50 and the maximum value is 50. The difference between the FROM and TO parameter values determines the number of files a user is able to select for this command. For example, if FROM(1) and TO(1) is specified then only 1 file may be selected. If FROM(1) and TO(50) is specified then 50 files may be selected by the user.

PHY_ONLY

Specifies whether only physical files will be presented for selection, or both physical and logical files.

SGL_ONLY

Specifies whether only 1 : 1 (single) relationships will be presented for selection, or both 1 : 1 and 1 : N (many) relationships.

PROMPT

Specifies the prompt text for the command. This is up to a maximum of 74 characters. The prompt text may contain special variables (refer to the [14.22.5 Special Template Variable Notes](#) on special variables). More detailed prompt text can be placed in the EXTEND parameter.

EXTEND

Specifies the extended prompt text for the command if the PROMPT parameter cannot contain a full enough description required prompt. This is up to a maximum of 8 lines of 74 characters. The extended prompt may contain special variables.

HELPIDS

Specifies up to 10 HELP panel identifiers for this application template. These will be displayed as full page screens of HELP when the user presses the HELP function key.

Examples

The following examples apply to the @@GET_FILS command.

Example 1: Ask the user to select a single primary physical file.

```
@@GET_FILS FROM(1) TO(1) PHY_ONLY(*YES) PROMPT('Select +  
the primary physical file to be worked with') +  
EXTEND('Enter the name of the PHYSICAL file ....') +  
HELPIDS(HELP010)
```

Example 2: Ask the user to select up to 20 related files. Both physical and logical files may be selected.

```
@@GET_FILS TO(20) PHY_ONLY(*NO) SGL_ONLY(*YES) PROMPT('Ei  
the name of the primary file to be used by this +  
template') +  
EXTEND('The file name may be specified in full, +  
partially .....') +  
HELPIDS(HELP020 HELP030 HELP040 HELP050)
```

14.6 @@GOTO Command

The @@GOTO command is used to pass control to a command label. The command label nominated must be associated with another command within the application template.

Required

```
@@GOTO -----LABEL----- command label -----
```

Parameters

LABEL

specifies the label of the command which will receive control. The label specified in this parameter must be the label of one and only one other command in the application template.

Examples

The following example applies to the @@GOTO command.

If none of a set of conditions is met then branch to a label:

```
@@IF COND((*IF @@CANS001 *EQ A)) GOTO(LB1)
@@IF COND((*IF @@CANS001 *EQ B)) GOTO(LB2)
@@GOTO LABEL(LB3)
```

.

LB1: @@LABEL

.

LB2: @@LABEL

.

LB3: @@LABEL

14.7 @@IF Command

The @@IF command is used to test the truth of a condition and then bypass the generation of certain RDML commands only if the condition is true.

The command label of the GOTO parameter must be used on another application template command which is the subject of the GOTO.

Required

```
@@IF ----- COND ----- *IF variable *EQ value -----  
>  
| *AND      *GT      |  
| *OR       *LT      |  
|           *NE      |  
|           *GE      |  
|           *LE      |  
|           |  
----- 40 max -----  
  
>---GOTO -----command label -----|
```

Parameters

COND

Specifies the condition to be evaluated to test the "truth" of the IF condition. The four parts of the parameter are the relationship (*IF *AND *OR), the variable to be evaluated (e.g. @@CANS001), the relational operator (*EQ *GT *LT *NE *GE *LE) and the literal value used for comparison (which must be of the same type as the variables being evaluated). The variable to be evaluated may be any application template variable.

GOTO

Specifies the label of the command which is to receive control. The label specified in this parameter must be the label of one and only one other command in the function.

Examples

The following examples apply to the @@IF command.

Example 1: If the answer to a question is "NO" then branch to a label to bypass generating some RDML code.

```
@@IF COND((*IF @@CANS001 *EQ NO)) GOTO(LB1)
```

```
Some RDML code
```

```
.  
. .  
.
```

```
LB1: @@LABEL
```

Example 2: If the user selected more than 1 file then ask the user to select additional fields.

```
@@IF COND((*IF @@TFMX *EQ 1)) GOTO(A25)
```

```
@@CLR_LST ....
```

```
@@MAK_LSTS ....
```

```
@@MRG_LSTS ....
```

```
A25: @@LABEL
```

14.8 @@INC_IDX Command

The @@INC_IDX command is used to increment an index by 1. This index name may be a new or existing index. If the index is a new index name, then the index value will be set to 1. The maximum number of indexes that can be used in an application template is 999. Required

```
@@INC_IDX ----- IDX_NAME ----- name -----
```

Parameters

IDX_NAME

Specifies the new or existing index name. The first character of the two character index name must be non-numeric.

Examples

The following example applies to the @@INC_IDX command.

Increment index CF.

```
@@INC_IDX IDX_NAME(CF)
```

14.9 @@LABEL Command

The @@LABEL command is used in conjunction with the other application template commands and specifies a label used to control the execution of application template commands and/or the generation of RDML code.

Refer to the @@IF, @@GOTO, @@CMP_IDX, etc. commands for more details and examples of this command.

```
@@LABEL ----- no parameters -----|
```

Examples

The following example applies to the @@LABEL command.

Example 1: If the answer to a question is "NO" then branch to a label to bypass generating some RDML code.

```
@@IF COND>(*IF @@CANS001 *EQ NO)) GOTO(LB1)
Some RDML code
```

```
·
·
·
```

```
LB1: @@LABEL
```

Example 2: If index value is greater than the highest file number selected, then finish building list of fields.

```
A10: @@LABEL
@@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) IS_GT(A20)
@@RTV_FLDS .....
@@INC_IDX .....
@@GOTO LABEL(A10)
A20: @@LABEL
```

14.10 @@MAK_LSTS Command

The @@MAK_LSTS command is used to make a list(s) from fields selected by the user. The list being built will be **automatically cleared** before executing this command.

The user will be presented with a field selection list (constructed from other nominated lists) from which all required fields may be selected. The method of both selection and pre-selection of fields is determined by the INTO_LSTS parameter. If the FORCE_LSTS parameter is used, then all fields in this list must be selected by the user.

Required

```
@@MAK_LSTS ---- FROM_LSTS ----- nn -----  
-->  
      |      |  
      -- 5 max --
```

Optional

```
>--- FORCE_LSTS ---- nn ----->
```

```
>--- INTO_LSTS -- nn -- col hdg 1 -- col hdg 2 ---->  
      |                               |  
      |                               |  
      | -- col hdg 3 -- *YESNO  -- *NO ---->  
      |           *SEQUENCE *ALL  |  
      |           *FORCE  |  
      |                               |  
      ----- 2 max -----
```

```
>--- HELPIDS ---- HELP panel identifiers -----|  
      |                               |  
      ----- 10 max -----
```

Parameters

FROM_LSTS

Specifies the list numbers from which the list of fields for selection will be built. These list numbers should have been previously built by the @@RTV_FLDS or @@MAK_LSTS commands. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes.

FORCE_LSTS

Specifies a list number (previously built by the @@RTV_FLDS or @@MAK_LSTS commands) which contains all fields to be selected by the user. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes.

INTO_LSTS

Specifies the lists to be made from the selected fields. This parameter consists of these parts:

- **List number** - specifies the list number to be built. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes. This list will be automatically cleared before executing this command.
- **Column heading 1** - specifies column heading line 1 to appear over this field selection column.
- **Column heading 2** - specifies column heading line 2 to appear over this field selection column.
- **Column heading 3** - specifies column heading line 3 to appear over this field selection column.
- **Field selection method** - must be either *YESNO or *SEQUENCE. This specifies the method the user can use to select the required fields. If *YESNO (the default) is chosen, then the user can select any fields by simply placing any non-blank character in the selection column. Pre-selected fields will have "Y" in the select column. If *SEQUENCE is chosen, then the user can sequence the order that the chosen fields are placed in the resulting list. This sequence number is a decimal 4,1 field. Pre-selected fields will have a sequence number incremented by 10 in the select column.

- **Pre-select fields** - must be either *NO, *ALL or *FORCE. This specifies whether fields are to be pre-selected in the list(s). If *NO (the default) is chosen, then none of the fields will be pre-selected. If *ALL is chosen, then all fields will be pre-selected. If *FORCE is chosen, then all fields that are in the FORCE_LSTS list will be pre-selected.

HELPIDS

Specifies up to 10 HELP panel identifiers for this application template. These will be displayed as full page screen of HELP when the user presses the HELP function key.

Examples

The following examples apply to the @@MAK_LSTS command.

Example 1: Ask the user to select fields that cannot be updated in the primary file. Fields can be selected by any non-blank character.

```
@@MAK_LSTS FROM_LSTS(2) INTO_LSTS((5 'Fields that' 'Cannot be'
    'Updated' *YESNO *NO)) HELPIDS(HELP010 HELP020)
```

Example 2: Ask the user to select fields that are to be displayed on the data entry panel, forcing the keys of the file to be selected. The fields can be ordered using a sequence number, and all fields are pre-selected.

```
@@MAK_LSTS FROM_LSTS(1) FORCE_LSTS(3) INTO_LSTS((4 'Fields t
    'Appear on' 'Add Panel' *SEQUENCE *ALL))
```

14.11 @@MRG_LSTS Command

The @@MRG_LSTS command is used to update a list by merging other lists with it, and can also optionally merge attributes for fields in the list. Attributes will only be merged if the field does not already exist in the INTO_LST.

Required

```
@@MRG_LSTS ---- FROM_LSTS ----- nn --- attributes -----  
--->  
      |      |      |      |  
      |      --- 7 max ---- |  
      |      |      |  
      ----- 2 max -----  
  
>--- INTO_LST ----- nn -----|
```

Parameters

FROM_LSTS

Specifies the list numbers from which the list is to be built. The optional attributes on each list specify the 10 character attributes to be merged with the fields in the FROM_LST if this field does not already exist in the INTO_LST. If the field is already in the INTO_LST then the attribute will not be merged. These list numbers should have been previously built by the @@RTV_FLDS or @@MAK_LSTS commands. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes.

INTO_LST

Specifies the list to be built from the list merging. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes.

Examples

The following examples apply to the @@MRG_LSTS command.

Example 1: Merge user selected fields which can't be updated in the primary file with the *OUTPUT attribute, with the display panel fields list.

```
@@CLR_LST NUMBER(6)  
@@MRG_LSTS FROM_LSTS((5 *OUTPUT)) INTO_LST(6)
```

Example 2: Merge key fields of the file with the *HIDDEN attribute into the display panel fields list.

```
@@RTV_KEYS OF_FILE(1) INTO_LST(3)  
@@CLR_LST NUMBER(4)  
@@MRG_LSTS FROM_LSTS((3 *HIDDEN)) INTO_LST(4)
```

14.12 @@QUESTION Command

The @@QUESTION command is used to ask a question and to receive a valid reply.

Required

```
@@QUESTION -----PROMPT----- 'text' ----->
>
```

```
>-----ANSWER ----- @@CANSnnn ----->
      @@NANSnnn
```

Optional

```
>-----EXTEND ----- 'text' ----->
      |           |
      -- 8 max --
```

```
>-----LOWER ----- *NO ----->
      *YES
```

```
>---- VALUES ----- compare value ----->
      |           |
      ----- 40 max -----
```

```
>---- RANGE ----- low value -- high value ----->
      |           |
      ----- 20 max -----
```

```
>---- SPCVAL --- from value -- replacement value -->
      |           |
      ----- 40 max -----
```

```
>-----HELPIDS ---- HELP panel identifiers -----|
      |           |
      ----- 10 max -----
```

Parameters

PROMPT

Specifies the prompt text for the question. This is up to a maximum of 74 characters. The prompt text may contain special variables (refer to the [14.22.5 Special Template Variable Notes](#) on special variables). More detailed prompt text can be placed in the EXTEND parameter.

ANSWER

Specifies the "special" variable that is to contain the answer to the question. It must be one of the special question and answer variables described in a following section (i.e. @@CANSnnn or @@NANSnnn). The last 2 characters of nnn may be a 2 character index name, which will be substituted by the current numeric value of the index.

EXTEND

Specifies the extended prompt text for the question if the PROMPT parameter cannot contain a full enough description of the question. This is up to a maximum of 8 lines of 74 characters. The extended text may contain special variables.

LOWER

Specifies whether the answer to the question is to remain in lowercase rather than being converted to uppercase.

VALUES

Specifies from 1 to 40 values to be checked against the answer.

RANGE

Specifies from 1 to 20 ranges of values to be checked against the answer. Each individual range must consist of a "low" value and a "high" value.

SPCVAL

Specifies from 1 to 20 values to be replaced by another value e.g. "Y" to "*YES".

HELPIDS

Specifies up to 10 HELP panel identifiers for this application template which will be displayed when the user presses the HELP function key. These are presented as full page screens of HELP.

Examples

The following examples apply to the @@QUESTION command.

Example 1: Ask the user if the change function key is to be enabled.

```
@@QUESTION PROMPT('Is the change function key to be +
enabled ?') +
ANSWER(@@CANS001) EXTEND('Reply Y or N only'....) +
LOWER(*NO) SPCVAL((Y *YES) ('y' *YES) (N *NO) +
('n' *NO)) HELPIDS(HELP010 HELP020)
```

Example 2: Ask the user how many conditions are to be tested.

```
@@QUESTION PROMPT('How many conditions are to be tested ?') +
ANSWER(@@NANS001) EXTEND('Indicate the number of +
conditions that are to be generated' 'by this +
application template') HELPIDS(HELP040 HELP050)
```

14.13 @@RTV_FLDS Command

The @@RTV_FLDS command is used to retrieve all the fields for the specified file number into the required list.

Note that the file must have already been chosen by the @@GET_FILS command, and the list must have been defined/cleared by the @@CLR_LST command.

```

                                     Required
@@RTV_FLDS ----- FROM_FILE --- number -----
----->
                                     index
>----- INTO_LST ---- number ----->
                                     index
-----
                                     Optional
>----- REAL_ONLY ---- *NO ----->
                                     *YES
>----- VIRT_ONLY ---- *NO ----->
                                     *YES
>----- ALPHA_ONLY --- *NO ----->
                                     *YES
>----- NUM_ONLY ----- *NO -----|
                                     *YES
```

Parameters

FROM_FILE

Specifies the file number for which the fields are to be retrieved. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to

the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes. Note that the file number must have already been selected by a @@GET_FILS command.

INTO_LST

Specifies the list number into which the fields are to be added. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes. Note that the list number must have been previously defined by an @@CLR_LST command.

REAL_ONLY

Specifies whether only "real" fields from the file should be retrieved into the list.

VIRT_ONLY

Specifies whether only "virtual" fields from the file should be retrieved into the list.

ALPHA_ONLY

Specifies whether only alphanumeric fields from the file should be retrieved into the list.

NUM_ONLY

Specifies whether only numeric fields from the file should be retrieved into the list.

Examples

The following examples apply to the @@RTV_FLDS command.

Example 1: Retrieve all fields from file 1 into list 1.

```
@@CLR_LST NUMBER(1)  
@@RTV_FLDS FROM_FILE(1) INTO_LST(1)
```

Example 2: Retrieve all "real" fields from file CF into list 2.

```
@@CLR_LST NUMBER(2)  
@@RTV_FLDS FROM_FILE(CF) INTO_LST(2) REAL_ONLY(*YES)
```

Example 3: Retrieve all numeric fields from file 3 into list 3.

@@CLR_LST NUMBER(3)

@@RTV_FLDS FROM_FILE(3) INTO_LST(3) NUM_ONLY(*YES)

14.14 @@RTV_KEYS Command

The @@RTV_KEYS command is used to retrieve all the key fields for the specified file number into the required list.

Note that the file must already have been chosen by the @@GET_FILS command, and the list must have been defined/cleared by the @@CLR_LST command.

Required

```
@@RTV_KEYS ----- OF_FILE ----- number -----  
-->  
                index  
  
                >----- INTO_LST ----- number -----|  
                index
```

Parameters

OF_FILE

Specifies the file number for which the key fields are to be retrieved. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes. Note that the file number must have already been selected by an @@GET_FILS command.

INTO_LST

Specifies the list number into which the fields are to be added. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes. Note that the list number must have been previously defined by an @@CLR_LST command.

Examples

The following examples apply to the @@RTV_KEYS command.

Example 1: Retrieve all key fields from file 1 into list 1.

```
@@CLR_LST NUMBER(1)  
@@RTV_KEYS OF_FILE(1) INTO_LST(1)
```

Example 2: Retrieve all key fields from file CF into list 2.

```
@@CLR_LST NUMBER(2)  
@@RTV_KEYS OF_FILE(CF) INTO_LST(2)
```

14.15 @@RTV_RELN Command

The @@RTV_RELN command is used to retrieve the relationship (the key fields of the access route) for the specified file number into the required list. This defines how this file is "related" to the file at the next higher level. This applies to all files except the primary file.

Note that the values returned would actually come from the "parent" or "joined" file, not the file nominated by the OF_FILE parameter.

Note that the file must already have been chosen by the @@GET_FILS command, and the list must have been defined/cleared by the @@CLR_LST command.

Required

```
@@RTV_RELN ----- OF_FILE ----- number -----  
-->  
           index  
  
>----- INTO_LST ----- number -----|  
           index
```

Parameters

OF_FILE

Specifies the file number for which the relationship (key fields in the access route) is to be retrieved. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes. Note that the file number must have already been selected by an @@GET_FILS command.

INTO_LST

Specifies the list number into which the fields are to be added. This number may be a 1 or 2 character numeric or a 2 character index name. If it is an index name, then the current index value will be substituted in this command. Refer to the @@SET_IDX, @@INC_IDX, @@DEC_IDX, @@CMP_IDX commands for setting and using indexes. Note that the list number must have been previously

defined by an @@CLR_LST command.

Examples

The following examples apply to the @@RTV_RELN command.

Example 1: Retrieve the relationship of file 2 into list 2.

```
@@CLR_LST NUMBER(2)
@@RTV_RELN OF_FILE(2) INTO_LST(2)
```

Example 2: Retrieve the relationship of file "CF" into list 3 and use this to FETCH the related record from file "CF".

```
@@CLR_LST NUMBER(3)
@@RTV_RELN OF_FILE(CF) INTO_LST(3)
FETCH   FIELDS(#PANELDATA) FROM_FILE(@@FNAMECF) WITH_I
```

14.16 @@SET_IDX Command

The @@SET_IDX command is used to set an index value to a numeric value. This index name may be a new or existing index. The maximum number of indexes which can be used in an application template is 999.

Required

```
@@SET_IDX ----- IDX_NAME ----- name -----  
->  
  
>----- TO ----- value -----|  
          numeric variable
```

Parameters

IDX_NAME

Specifies the new or existing index name. The first character of the two character index name must be non-numeric.

TO

Specifies the numeric value to which the index is to be set. This may be any valid numeric variable or any valid numeric literal.

Examples

The following examples apply to the @@SET_IDX command.

Example 1: Set index AB to 2.

```
@@SET_IDX IDX_NAME(AB) TO(2)
```

Example 2: Set index CF to the maximum file number selected.

```
@@SET_IDX IDX_NAME(CF) TO(@@TFMX)
```

14.17 General Template Variables

14.22.5 Special Template Variable Notes

14.22 Tips for Template Programming

Variable	Description	Type	Len	Dec
@@COMPANY	Name of current company / organization	A	30	
@@DATE	Date in installation format (xx/xx/xx)	A	8	
@@DATE8	Date in installation format (xx/xx/xxxx or xxxx/xx/xx)	A	10	
@@DECIMAL	Decimal format ('.' or ',')	A	1	
@@FUNCDES	Current LANSAs function description	A	40	
@@FUNCTION	Current LANSAs function name	A	7	
@@GENNAME/xx/yy/zzzzzz	Generate field names	A	Variable	
@@INDEXii	Numeric value of index ii	N	2	0
@@JOBNAME	Current IBM i job name	A	10	
@@JOBNBR	Current IBM i job number	A	6	
@@PRODREL	Current LANSAs release level	A	4	
@@PROCDES	Current LANSAs process description	A	40	
@@PROCESS	Current LANSAs process name	A	10	
@@PRODUCT	Product name (i.e.: LANSAs)	A	5	
@@TIME	Current time (xx:xx:xx)	A	8	
@@USER	Current IBM i user identity	A	10	

Note:

- ii is a valid index name of 2 characters. The numeric value will be substituted for the variable.
- xx is a number from 1 to 99. This length is the maximum length. The number xx may be specified as 1 or 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).
- yy is a number from 1 to 99. This length is the maximum length. The number yy may be specified as 1 or 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).
- zzzzzz is a character name of length 1 to 6. It must be followed by a blank when used in a template.
- @@GENNAME/xx/yy/zzzzzz will be expanded into a list of elements prefixed by zzzzzz and suffixed by xx and from 1 to yy.

For example:

```
DEF_LINE NAME(#LINEDATA) FIELDS(@@GENNAME/01/06/#TOT
```

would result in this RDML code in a template:

```
DEF_LINE NAME(#LINEDATA) FIELDS( #TOT0101 #TOT0102  
#TOT0103 #TOT0104 #TOT0105 #TOT0106 )
```

14.18 Question and Answer Template Variables

[14.22.5 Special Template Variable Notes](#)

[14.22 Tips for Template Programming](#)

Variable	Description	Type	Length	Dec
@@CANSnnn	Character answer	A	74	
@@NANSnnn	Numeric answer	N	15	5

where nnn is a number from 001 to 999.

The length is a maximum length. This number may be specified as 3 numerics, or 1 numeric followed by a valid index name of 2 characters (its numeric value will be substituted in the variable name).

14.19 File Template Variables

14.22.5 Special Template Variable Notes

14.22 Tips for Template Programming

Variable	Description	Type	Len	Dec
@@FNAMEnn	File name	A	10	
@@FLIBRnn	File library	A	10	
@@FVERSnn	File version number	N	15	5
@@FTYPEnn	File type (P=PF, L=LF)	A	1	
@@FDESCnn	File description	A	40	
@@FBASPnn	Based in physical file name	A	10	
@@FRELFnn	Related file name	A	10	
@@FRELLnn	Related file library	A	10	
@@FRELVnn	Related file version number	N	15	5
@@FRELRnn	Related file relationship (O=1:1, M=1:n)	A	1	
@@FRELAnn	Related file access route name	A	10	
@@FRELCnn	Related file connection type. The connection types are: BASE = Base file DIRBASE = Directly connected to the base file INDBASE = Indirectly connected to the base file DETAIL = Detail file (i.e. 1 : Many related file) DIRDETL = Directly connected to detail file INDDETL = Indirectly connected to detail file	A	70	
@@FAREAnn	Header or browse area (H=HDR,B=BRW)	A	1	
@@TFMX	Maximum file number selected	N	15	5
@@TFMN	Minimum file number selected	N	15	5

where nn is a number from 1 to 99. This length is the maximum length. The number nn may be specified as 1 or 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).

14.20 List Template Variables

14.22.5 Special Template Variable Notes

14.22 Tips for Template Programming

See Note	Variable	Description	Type	Len	Dec
1	@@LSTnn	List name - all elements of list	A	Variable	
1	@@LSUnn	List name - all elements of list	A	Variable	
2	@@LSXnn/yy	List name - first yy elements of list	A	Variable	
1	@@LNEnn	Number of elements in list nn	N	2	0
3	@@LELnnxx	Element xx of list nn	A	10	
3	@@LATnnxx	Attributes of element xx of list nn	A(7)	10	
3	@@LDSnnxx	Description of element xx of list nn	A	40	
3	@@LTPnnxx	Type of element xx of list nn (A,P,S)	A	40	

Note 1

nn is a number from 1 to 99. This length is the maximum length. The number nn may be specified as 1 or 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).

@@LSTnn will be expanded to include all elements (preceded by a "#" if a non-literal) with their corresponding attributes.

For example:

```
GROUP_BY NAME(#PANELDATA) FIELDS(@@LST01) in the template  
may be substituted for in the resulting RDML code by  
GROUP_BY NAME(#PANELDATA) FIELDS((#EMPNO *OUTPUT) #SU  
#ADDR1 #ADDR2)
```

@@LSUnn will be expanded to include all elements with no preceding "#" and no attributes.

Note 2

nn is a number from 1 to 99. This length is the maximum length. The number nn may be specified as 1 or 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).

yy is a number from 1 to 99. This length is the maximum length. The number yy may be specified as 1 or 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).

@@LSXnn/yy will be expanded to include the first yy elements (preceded by a "#" if a non-literal) of the list.

For example:

```
GROUP_BY NAME(#PANELDATA) FIELDS(@@LSX01/03)
```

would result in this RDML code in a template:

```
GROUP_BY NAME(#PANELDATA) FIELDS(#EMPNO #SURNAME #ADI
```

Note 3

nn is a number from 1 to 99. This length is the maximum length. The number nn may be specified as 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).

xx is a number from 1 to 99. This length is the maximum length. The number xx may be specified as 1 or 2 numerics, or a valid index name of 2 characters may be used (and its numeric value will be substituted in the variable name).

14.21 Template Error Messages

The execution of certain Application Template Commands may result in error/warning messages being displayed. For further information refer to the [Error Message Notes](#). The error will be one of these:

Error Commands Code	Meaning
2 @@IF @@CMP_IDX	Command label specified on GOTO parameter is not found in this application template.
3 @@GOTO	Command label specified is not found in this application template.
4 @@CLR_LST	Index specified for list number is invalid.
5 @@SET_IDX @@INC_IDX @@DEC_IDX @@CMP_IDX	Only 999 indexes can be specified in an application template.
6 @@SET_IDX @@CMP_IDX	Invalid numeric value. The literal is not a valid numeric or the special @@ variable is unknown in this template.
7 @@RTV_FLDS @@RTV_KEYS @@RTV_RELN	Index specified for file number is invalid.
8 @@RTV_FLDS @@RTV_KEYS @@RTV_RELN	File number specified is not known to this application template.
9 @@RTV_FLDS @@RTV_KEYS @@RTV_RELN	Index specified for list number is invalid.
10 @@RTV_FLDS @@RTV_KEYS @@RTV_RELN @@MRG_LSTS	List number specified is not known to this application template. An @@CLR_LST command has not been previously executed for this list number.

11	@@MRG_LSTS	Index specified for "into" list number is invalid.
12	@@MRG_LSTS	Index specified for "from" list number is invalid.
13	@@MAK_LSTS	Index specified for force list number is invalid.
14	@@MAK_LSTS	Index specified for "from" list number is invalid.
15	@@MAK_LSTS	Index specified for "to" list number is invalid.
16	RDML cmds	Index specified in @@INDEXii variable is invalid.

Error Message Notes

These messages will appear on the display of the application template command being executed e.g. DCM0793 is issued for an @@QUESTION message if the answer is not a valid reply according to the parameters on the @@QUESTION command.

If a fatal error occurs in the execution of an application template, then all RDML code generated by the application template so far will be backed out, and the execution of the application template will terminate. Error message DCM0794:

"Error occurred in application template tttttttt at sequence no. nnnnn.nn - error code xxx"

will appear on the next LANSAs screen to write out the message subfile. The sequence number in the application template is the **command sequence number** rather than the line number.

14.22 Tips for Template Programming

14.22.1 [Accepting Errors On Commands](#)

14.22.2 [Forcing EDTSRC/SEU to Update With Errors](#)

14.22.3 [The Double Bracket Trap](#)

14.22.4 [Testing with an Alternate Session](#)

14.22.5 [Special Template Variable Notes](#)

14.22.1 Accepting Errors On Commands

Note that some RDML commands will not be accepted by the IBM CL syntax checker when they contain special variables such as @@LSTnn. They must be **forced to accept the commands**. An example of this is the following:

```
GROUP_BY NAME(#PANELDATA) FIELDS(@@LST03)
```

After executing an application template containing this RDML command, and assuming list number 3 contains selected fields, RDML commands like this will be generated:

```
GROUP_BY NAME(#PANELDATA)
      FIELDS((#EMPNO *OUTPUT) #SURNAME
      #ADDRESS1 #ADDRESS2)
```

which is quite valid.

Take care not to code the original RDML command in the application template as in the following example:

```
GROUP_BY NAME(#PANELDATA) FIELDS((#@@LST03))
```

(which will be accepted by the command prompter) or

```
GROUP_BY NAME(#PANELDATA) FIELDS(((#@@LST03)))
```

(which is the automatic result when the command prompting is canceled) as both these examples will generate invalid RDML commands.

It is absolutely **essential** that all application templates which include RDML commands with special variables are thoroughly tested to ensure that the correct RDML command will be generated by the application template.

14.22.2 Forcing EDTSRC/SEU to Update With Errors

If your application template contains RDML commands that were forced to be accepted (as in point 1 above), then on exiting the EDTSRC/SEU editor, it will also be necessary to force the update with errors. This is necessary to ensure the application template will generate correct RDML code.

14.22.3 The Double Bracket Trap

If a command is forced to be accepted (as described above), it will often be necessary to further alter the command to ensure that the correct RDML command will be generated by the template. For example, if this command is required:

```
GROUP_BY NAME(#PANELDATA) FIELDS(@@LST03)
```

If this command is **forced to be accepted** by canceling the command, the resulting command will look like this:

```
GROUP_BY NAME(#PANELDATA) FIELDS(((@@LST03)))
```

After executing an application template containing this RDML command, and assuming list number 3 contains selected fields, an **invalid RDML command** will be generated as follows:

```
GROUP_BY NAME(#PANELDATA) FIELDS(((#EMPNO *OUTPUT)
#SURNAME #ADDRESS1 #ADDRESS2))
```

or

```
GROUP_BY NAME(#PANELDATA) FIELDS((#EMPNO #SURNAME
#ADDRESS1 #ADDRESS2))
```

14.22.4 Testing with an Alternate Session

To aid in testing application templates, it is useful to use one session for coding and modifying the application template (from the Housekeeping Menu), and an alternate session to actually execute the template (from the Process Control Menu).

14.22.5 Special Template Variable Notes

Application templates may contain **special variables**. These special variables, used in both Application Templates Command and RDML Commands, will be replaced by their corresponding value(s). These values may be derived as the result of other Application Template commands. For example, @@QUESTION will create a variable @@CANSnnn or @@NANSnnn which may be subsequently used in an @@IF command or an @@COMMENT command; @@GET_FILS will create file variables; @@MAK_LSTS will create list variables; etc.

Note that some RDML commands **will not be accepted** by the IBM CL syntax checker when they contain these special variables. They must be **forced** to accept the commands, and the **SEU editor** must also be forced to update with errors. An example of this is:

```
GROUP_BY NAME(#PANELDATA) FIELDS(@@LST03)
```

After executing an application template containing this RDML command, and assuming list number 3 contains selected fields, an RDML command will be generated as follows:

```
GROUP_BY NAME(#PANELDATA) FIELDS((#EMPNO *OUTPUT) #SURNAME  
#ADDRESS1 #ADDRESS2)
```

which is quite valid.

Take care not to code the original RDML command in the application template as in the following example:

```
GROUP_BY NAME(#PANELDATA) FIELDS((#@@LST03))
```

as this will generate an invalid RDML command.

It is absolutely **essential** that all application templates, which include special variables contained in RDML commands are thoroughly tested to ensure that the correct RDML command will be generated by the application template.

14.23 Sample Application Templates

The following are provided as **examples only** of application templates. You should read and understand these examples before attempting to write/modify application templates yourself.

[14.23.1 Simple data entry program](#)

[14.23.2 Header/Detail style inquiry program](#)

14.23.1 Simple data entry program

1. The following is an example of a **simple** application template for a data entry program:

```
/* ===== */
/* ASK FOR THE "WORD" - ask the user a question.      */
/* Note the prompt text and extended prompt text that */
/* can be entered on the command. More detailed help  */
/* can also be entered in HELP panels for the template.*/
/* Note the special format of the ANSWER parameter.  */
/* ===== */
@@QUESTION PROMPT('Supply word to describe+
  WHAT this data entry program wo+
  rks with') ANSWER(@@CANS001) EX+
  TEND('The word you specify here+
  is used to build messages that +
  appear on the 'data entry scre+
  en panel. You should use ONE wo+
  rd only, use lowercase' 'charac+
  ters only and only use singular+
  form (eg: "customer", "employee+
  "" "order"). Do NOT use more t+
  han 18 characters in your answe+
  r.' 'Use the HELP function key +
  for more information and exampl+
  es.') HELPIDS (HELP010 HELP020 +
  HELP030 HELP040 HELP050)
/* =====*/
/* GET NAME OF JUST ONE PHYSICAL FILE                */
/* Ask the user to enter the name of a single primary */
/* file used for data entry. Note the prompt text    */
/* and extended prompt text that can be entered on the */
/* command. More detailed help can also be entered on */
/* HELP panels for the template.                    */
/* =====*/
@@GET_FILS TO(1) PROMPT('Enter name of PHYSICAL
  file to be used by this template') E+
  XTEND('The file name may be specifie+
```

d partially (to cause a partial 'list of available files to be displayed), or in left blank (to cause a full list of available files to be displayed). When a list of files is displayed, the file required may be selected from the list. Use the HELP function key for more details about this template and examples of the type of RDML applications it can create.) HELPIDS(HELP010 HELP020 HELP030 HELP040 HELP050)

```

/* ===== */
/* GET FIELDS OF CHOSEN FILE INTO LIST 1      */
/* Note that all lists must be defined by an  */
/* @@CLR_LST command before being used in an  */
/* application template.                      */
/* ===== */
@@CLR_LST NUMBER(1)
@@RTV_FLDS FROM_FILE(1) INTO_LST(1)
/* ===== */
/* GET KEYS OF CHOSEN FILE INTO LIST 2      */
/* ===== */
@@CLR_LST NUMBER(2)
@@RTV_KEYS OF_FILE(1) INTO_LST(2)
/* ===== */
/* GET USER TO CHOOSE FIELDS TO APPEAR ON PANEL */
/* AND PUT RESULTS INTO LIST 3              */
/* Note how the keys of the file are used as a */
/* force list to ensure all the fields are chosen. */
/* Note the column headings for the selection column */
/* and that sequence numbers are pre-filled on all */
/* fields in the selection list. This allows fields */
/* to be ordered in the desired sequence.      */
/* ===== */
@@CLR_LST NUMBER(3)
@@MAK_LSTS FROM_LSTS(1) FORCE_LSTS(2) INTO_LSTS+
((3 'Fields to' 'Appear on' 'Entry P+
anel' *SEQUENCE *ALL)) HELPIDS(HELP0+

```

```

10 HELP020 HELP030 HELP040 HELP050)
/* ===== */
/* GET USER TO CHOOSE FIELDS TO WORK WITHIN      */
/* PROGRAM AND PUT RESULTS INTO LIST 4           */
/* Note that this list only requires the user to */
/* enter anynon-blank character to select a field.*/
/* =====*/
@@CLR_LST NUMBER(4)
@@MAK_LSTS FROM_LSTS(1) INTO_LSTS((4 'Fields to '+'
Work with "in Program ' *YESNO *NO)) H+
ELPIDS(HELP010 HELP020 HELP030 HELP040 H+
ELP050)
/* ===== */
/* MERGE FIELDS IN LIST 4 INTO LIST 3 AS *HIDDEN */
/* The fields selected as fields to be worked with in */
/* the program are merged to list 3 with the *HIDDEN */
/* attribute if not already selected in list 3 by the */
/* previous @@MAK_LSTS command.                      */
/* ===== */
@@MRG_LSTS FROM_LSTS((4 *HIDDEN)) INTO_LST(3)
/* ===== */
/* ASK HOW THE PANEL IS TO BE DESIGNED           */
/* ===== */
@@QUESTION PROMPT('Design fields on data entry+
panel DOWN the screen or ACROSS the+
screen') ANSWER(@@CANS002) EXTEND('+
Reply DOWN or ACROSS only.' 'If you+
r data entry panel contains 17 (or +
less) fields, DOWN is the 'reco+
mmended value. If your data entry p+
anel contains more than 17' 'fields+
, ACROSS is the recommended value.'+
'Use the HELP function key for more+
information and examples.') LOWER(*+
NO) VALUES(DOWN ACROSS) HELPIDS(HEL+
P010 HELP020 HELP030 HELP040 HELP05+
0)
/* =====*/
/* GENERATE THE RDML PROGRAM                      */

```

```

/* The following code consists only of RDML that */
/* will appear in the generated RDML program. */
/* Note the use of special @@ variables in the RDML */
/* commands--these are substituted when the template */
/* is executed. */
/* ===== */
@@COMMENT 'Function control options'
FUNCTION OPTIONS(*NOMESSAGES *DEFERWRITE)
@@COMMENT 'Group and field declarations'
/* */
/* The following command will not be accepted by the */
/* CL syntax checker, it must be forced to be accepted */
/* as it will be quite valid when the special variable */
/* @@LST03 is relaxed by the list 3 elements when */
/* executing this template. Do not code */
/* FIELDS((#@LST03))as this will generate */
/* an invalid RDML command. This error */
/* may also be true for other RDML commands. It */
/* will be necessary to force these errors to */
/* be accepted also (e.g. the DESIGN parameter of */
/* the REQUEST command) */
/* */
GROUP_BY NAME(#PANELDATA) FIELDS(@@LST03)
@@COMMENT 'Issue initial data entry message'
MESSAGE MSGID(DCU0010) MSGF(DC@M01) +
      MSGDTA("@@CANS001")
@@COMMENT 'Do data entry until terminated by +
      EXIT or CANCEL'
BEGIN_LOOP
@@COMMENT 'Request user inputs or corrects details'
REQUEST FIELDS(#PANELDATA) DESIGN(*@@CANS002)+
      IDENTIFY(*LABEL)
@@COMMENT 'Perform any program level validation here'
BEGINCHECK
ENDCHECK
@@COMMENT 'Attempt to insert data into the data base'
INSERT FIELDS((#PANELDATA)) TO_FILE(@@FNAME01)
@@COMMENT 'If okay, reset fields and issue accepted +
      message'

```

```
CHANGE FIELD(#PANELDATA) TO(*DEFAULT)
MESSAGE MSGID(DCU0011) MSGF(DC@M01) +
      MSGDTA("@@CANS001")
END_LOOP
/* ===== */
/* CLEAR ALL LISTS USED */
/* At the end of all application templates, it */
/* is suggested that all work lists in the */
/* template are cleared to delete all work records */
/* from the database. */
/* ===== */
@@CLR_LST NUMBER(1)
LST NUMBER(2)
@@CLR_LST NUMBER(3)
@@CLR_LST NUMBER(4)
```

14.23.2 Header/Detail style inquiry program

The following is an example of a **fairly complex** application template for a header/detail style inquiry program:

```
/* =====
/* GET NAMES OF UP TO 50 RELATED FILES          */
/* Note that the user can select up to 50 physical or */
/* logical files including 1:n relationships.      */
/* =====
@@GET_FILS TO(50) PHY_ONLY(*NO) SGL_ONLY(*NO)+
  PROMPT('Enter the name of the base+
  file to be used by this template')+
  EXTEND('The file name may be speci+
  fied partially (to cause a partia+
  l' 'list of available files to be +
  displayed), or in left blank (to c+
  ause a full list' 'of available fi+
  les to be displayed). When a list+
  of files is displayed,' 'the file +
  required may be selected from the +
  list.' ' 'Use the HELP function +
  key for more details about this te+
  mplate and' 'examples of the type+
  of RDML applications it can create+
  .') HELPIDS(HELP010 HELP020 HELP03+
  0 HELP040)
/* =====
/* LOAD DETAILS OF FIELDS OF "HEADER" INTO LIST 1      */
/* LOAD DETAILS OF FIELDS OF "BROWSE" INTO LIST 2      */
/* Use special variable @@FAREAnn to separate fields in */
/* the header and browse portions of the panel.        */
/* Note the use of an index to control the loading of  */
/* multiple file information.                          */
/* =====
  @@CLR_LST NUMBER(1)
  @@CLR_LST NUMBER(2)
  @@SET_IDX IDX_NAME(CF) TO(1)
A10: @@LABEL
```

```

@@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) IF_GT(A20)
@@IF COND((*IF @@FAREACF *NE B)) GOTO(A12)
@@RTV_FLDS FROM_FILE(CF) INTO_LST(2)
@@GOTO LABEL(A14)
A12: @@RTV_FLDS FROM_FILE(CF) INTO_LST(1)
A14: @@INC_IDX IDX_NAME(CF)
@@GOTO LABEL(A10)
A20: @@LABEL
/* =====*/
/* ASK THE USER TO SELECT THE HEADER FIELDS REQUIRED */
/* =====*/
@@CLR_LST NUMBER(11)
@@MAK_LSTS FROM_LSTS(1) INTO_LSTS((11 'Fields in'+
'Header' 'Area' *SEQUENCE *ALL)) HELPI+
DS(HELP010 HELP020 HELP030 HELP040)
/* =====*/
/* ASK THE USER TO SELECT THE BROWSE FIELDS REQUIRED */
/* =====*/
@@CLR_LST NUMBER(22)
@@MAK_LSTS FROM_LSTS(2) INTO_LSTS((22 'Fields in'+
'Detail/List' 'Area' *SEQUENCE *ALL)) +
HELPIDS(HELP010 HELP020 HELP030 HELPO +
40)
/* =====*/
/* ASK THE USER HOW TO DESIGN THE PANELS */
/* =====*/
@@QUESTION PROMPT('Design fields in the header a +
rea DOWN the screen or ACROSS the scre+
en') ANSWER(@@CANS002) EXTEND('Reply D+
OWN or ACROSS only.' 'If your header a+
rea contains 10 (or less) fields, DOWN+
is the ' 'recommended value.+
If your header area contains more than+
10' 'fields, ACROSS is the recommended+
value.' 'Use the HELP function key for+
more information and examples.') LOWER+
(*NO) VALUES(DOWN ACROSS) HELPIDS(HELP+
010 HELP020 HELP030 HELP040)
/* =====

```

```

/* MERGE ALL RELATED KEY FIELDS INTO LIST 11 OR LIST 22 */
/* AS *HIDDEN FIELDS. LIST 3 IS A WORKING LIST ONLY */
/* Note the use of @@RTV_RELN command to get the keys of */
/* the secondary files. */
/* =====
  @@SET_IDX IDX_NAME(CF) TO(2)
A30: @@LABEL
  @@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) +
  IF_GT(A40)
  @@CLR_LST NUMBER(3)
  @@RTV_RELN OF_FILE(CF) INTO_LST(3)
  @@IF COND((*IF @@FAREACF *NE B)) +
  GOTO(A34)
  @@IF COND((*IF @@FRELRCF *EQ M)) +
  GOTO(A34)
  @@MRG_LSTS FROM_LSTS((3 *HIDDEN)) INTO_LST(22)
  @@GOTO LABEL(A36)
A34: @@MRG_LSTS FROM_LSTS((3 *HIDDEN)) INTO_LST(11)
A36: @@INC_IDX IDX_NAME(CF)
  @@GOTO LABEL(A30)
A40: @@LABEL
/* =====
/* GENERATION OF RDML CODE STARTS HERE */
/* =====
  FUNCTION OPTIONS(*NOMESSAGES *DEFERWRITE)
  GROUP_BY NAME(#HEADER) FIELDS(@@LST11)
  DEF_LIST NAME(#LIST)
  FIELDS((#LISTDUMMY *HIDDEN) @@LST22)
  @@COMMENT 'Loop until user EXITs or CANCELs'
  BEGIN_LOOP
/* =====
/* REQUEST KEYS OF THE BASE FILE BE INPUT AND GET DATA */
/* =====
  @@CLR_LST NUMBER(3)
  @@RTV_KEYS OF_FILE(1) INTO_LST(3)
R10: REQUEST FIELDS(@@LST03) DESIGN(*@@CANS002) +
  IDENTIFY(*LABEL)
/* =====
/* GENERATE FETCH TO THE PRIMARY FILE */

```

```

/* =====
  @@COMMENT COMMENT('Fetch file @@FNAME01 details  ')
  FETCH  FIELDS((#HEADER)) +
    FROM_FILE(@@FNAME01) +
    WITH_KEY(@@LST03) NOT_FOUND(R10) +
    ISSUE_MSG(*YES)
/* =====
/* GENERATE FETCHES TO ALL FILES IN THE HEADER AREA  */
/* =====
  @@SET_IDX IDX_NAME(CF) TO(2)
H10: @@LABEL
  @@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) +
    IF_GT(H20)
  @@IF COND((*IF @@FAREACF *EQ B)) GOTO(H15)
  @@CLR_LST NUMBER(3)
  @@RTV_RELN OF_FILE(CF) INTO_LST(3)
  @@COMMENT COMMENT('Fetch file @@FNAMECF details  ')
  FETCH  FIELDS((#HEADER)) FROM_FILE(@@FNAMECF) +
    WITH_KEY(@@LST03)
H15: @@INC_IDX IDX_NAME(CF)
  @@GOTO LABEL(H10)
H20: @@LABEL
/* =====
/* NOW EXTRACT DATA TO BE PLACED INTO THE BROWSE LIST
/* =====
  @@SET_IDX IDX_NAME(CF) TO(2)
  @@SET_IDX IDX_NAME(SC) TO(0)
A50: @@LABEL
  @@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) +
    IF_GT(A80)
  @@IF COND((*IF @@FAREACF *NE B)) GOTO(A78)
  @@CLR_LST NUMBER(3)
  @@RTV_RELN OF_FILE(CF) INTO_LST(3)
  @@IF COND((*IF @@FRELRCF *EQ M)) GOTO(A55)
/* FETCH INTO THE LIST ENTRY */
  @@COMMENT COMMENT('Fetch file @@FNAMECF details  ')
  FETCH  FIELDS((#LIST)) FROM_FILE(@@FNAMECF) +
    WITH_KEY(@@LST03)
  @@GOTO LABEL(A78)

```

```

/* THE ONE AND ONLY SELECT COMMAND */
A55: @@COMMENT COMMENT('Select all file @@FNAMECF details')
    @@INC_IDX IDX_NAME(SC)
    SELECT  FIELDS((#LIST)) FROM_FILE(@@FNAMECF) +
           WITH_KEY(@@LST03)
    @@GOTO LABEL(A78)
/* INC INDEX AND LOOP AROUND */
A78: @@INC_IDX IDX_NAME(CF)
    @@GOTO LABEL(A50)
A80: @@LABEL
/* =====
/* ADD_ENTRY AND ENDSELECT FOR THE LIST (IF SELECT USED)
/* =====
    @@CMP_IDX IDX_NAME(SC) IDX_VALUE(0) IF_EQ(A90)
    ADD_ENTRY TO_LIST(#LIST)
    ENDSELECT
A90: @@LABEL
/* =====
/* DISPLAY DETAILS TO THE USER */
/* =====
    @@COMMENT COMMENT('Display results to the user')
    DISPLAY  FIELDS(#HEADER) DESIGN(*@@CANS002)+
           IDENTIFY(*LABEL)+
           BROWSELIST(#LIST)
    @@COMMENT COMMENT('Clear header and list and +
           loop around ')
    CHANGE  FIELD(#HEADER) TO(*DEFAULT)
    @@CMP_IDX IDX_NAME(SC) IDX_VALUE(0) IF_EQ(A95)
    CLR_LIST NAMED(#LIST)
A95: @@LABEL
    END_LOOP
/* =====
/* CLEAR ALL LISTS USED */
/* =====
    @@CLR_LST NUMBER(1)
    @@CLR_LST NUMBER(2)
    @@CLR_LST NUMBER(3)
    @@CLR_LST NUMBER(11)
    @@CLR_LST NUMBER(12)

```

15. External Resource Definitions

[15.1 External Resource Name](#)

[15.4 External Resource Description](#)

[15.6 External Resource Encoding](#)

[15.2 External Resource LANSAs Folder](#)

[15.3 External Resource File Name](#)

[15.5 External Resource Content Type](#)

External Resources allow you to manage files that are created or updated externally but are part of your LANSAs application. External resources may be Javascript, HTML-pages and images for web-development, or even simple configuration files for desktop-applications.

These files can be registered in the Repository, checked into an IBM i master and deployed using the LANSAs Deployment Tool.

Also see

[Register Multiple External Resources](#)

[Register a Single External Resource](#) in the *User Guide*.

15.1 External Resource Name

Mandatory.

Specify the name of the External Resource to be stored in the LANSAs Repository. This is a LANSAs name and will be used to identify the External Resource in the LANSAs Repository.

Rules

- Must be a valid [LANSAs Object Name](#).

Tips & Techniques

- Using the *Register External Resources* dialog you can quickly register many external resources at one time and they will be named consistently using your specified prefix.

Also See

[Create External Resource](#) in the *User Guide*

[↑ 15. External Resource Definitions](#)

15.2 External Resource LANSa Folder

Mandatory.

Specify the LANSa folder that is the root of External Resource location. Registering an External Resource in the context of a known LANSa folder allows you to move the External Resource from one system to another (Check-in/Check-out, Import/Export, Deployment) as all LANSa systems understand the physical location of their (logical) folders.

Tips & Techniques

- Whenever you select a file, its path is analyzed and the LANSa folder is automatically determined.

Also See

[Create External Resources](#) in the *User Guide*

[↑ 15. External Resource Definitions](#)

15.3 External Resource File Name

Mandatory.

Specify the file that is to be managed as an External Resource. This file will be stored in the LANSAs Repository and can be deployed to other LANSAs systems.

Tips & Techniques

- Select the LANSAs folder first, then the prompter will then take you to the corresponding directory.

Also See

[Create External Resources](#) in the *User Guide*

[↑ 15. External Resource Definitions](#)

15.4 External Resource Description

Mandatory.

Specify the description associated with the External Resource.

Tips & Techniques

- Select the file first, and the description will default to the file name.

Also See

[Create External Resources](#) in the *User Guide*

[↑ 15. External Resource Definitions](#)

15.5 External Resource Content Type

Optional.

Specifies whether an External Resource contains text or binary data. For text data you must specify the encoding.

Note: The encoding information is currently used only to provide the correct CCSID when extracting on an IBM i.

Also See

[Create External Resources](#) in the *User Guide*

[↑ 15. External Resource Definitions](#)

15.6 External Resource Encoding

Optional.

Specifies the text encoding of an External Resource contains that text data.

Note: The encoding information is currently used only to provide the correct CCSID when extracting on an IBM i.

Also See

[Create External Resources](#) in the *User Guide*

[↑ 15. External Resource Definitions](#)

16. Windows and Linux Considerations

16.1 Reporting Considerations

16.2 "Job" Numbers

16.3 Batch Jobs

16.4 IBM i Job Queue Emulation

16.5 The RUNSQL Utility

16.5 The RUNSQL Utility

16.7 Sizing RDML Windows

16.8 Windows 64-bit Support

16.9 Linux Differences

16.10 Code Page Considerations

16.11 Regional Settings

16.1 Reporting Considerations

When working with reporting functions with Visual LANSA you should be aware of the following:

- Output for reports will be directed to the printer port specified by the X_RUN parameter PRTR unless the special value PRTR=*PATH is specified.
- If the special value PRTR=*PATH is specified output for reports will be directed to files in the directory specified by the X_RUN parameter PPTH.

The name of each report file will be:
ffffff.nnn

where

ffffff is the function name (in a valid filename form)

nnn is the next consecutive number for this function in this directory. There can be a maximum of 999 report files in the PPTH directory for any one function at any time.

For example, a reporting function named FREPORT might output three reports. Executing this function would produce files FREPORT.001, FREPORT.002 and FREPORT.003 in the PPTH directory. If the same function is executed again it would produce files FREPORT.004, FREPORT.005 and FREPORT.006, and so on until previous FREPORT.nnn files are deleted.

Refer to the topic [Using the X_RUN Command](#) for details of the X_RUN parameters PRTR and PPTH.

16.2 "Job" Numbers

When using IBM i, a job number is automatically assigned by the system when a user signs onto a workstation or submits a job. At a workstation, multiple processes and functions can be run in separate invocations from the same job all having the same job number.

When running Visual LANSA applications, a job number will be assigned for each invocation of the X_RUN function whether it is run from the same window or not.

This information is important if your functions rely upon job numbers or use the job number system variable.

16.3 Batch Jobs

A batch job in Visual LANSAs is not the same as a batch job on the IBM i even though Visual LANSAs supports multi-tasking.

When a batch job in Visual LANSAs is executed, an independent session is started. This will appear as a child window in your application. The window will display the status of the batch job. All messages issued by the batch job will be displayed in the window. When the job completes, a message is displayed in the window.

If the batch job completes successfully, then the window will close automatically. If the batch job fails, a message will appear and the window will not disappear. It is necessary to click on OK to close the window.

Be careful when testing your executing batch jobs where you require single threaded job queues, i.e. batch jobs which must execute consecutively. Refer to [IBM i Job Queue Emulation](#) for details of how to execute batch jobs consecutively.

Batch Jobs on Linux

LANSAs only ever uses Linux as a Server so there is no interactive user interface provided for Linux. When executing on a Linux platform all batch messages are directed to the standard error file (stderr) and also to the system log.

You may redirect stderr to a file by appending **2> pathname**, where pathname is a filename, to your X_RUN command.

Refer to the *syslog man page* on your Linux system for details on how to capture LANSAs syslog events.

IBM i job queues can be emulated on Linux as described in [IBM i Job Queue Emulation](#).

16.4 IBM i Job Queue Emulation

A facility exists within Visual LANSA to allow Windows applications to emulate the type of processing, that can be provided by IBM i job queues and subsystems.

A reasonable understanding of IBM i job queues and subsystems is assumed knowledge throughout this section.

The LANSA RDML SUBMIT command is used to initiate a "batch" job from an executing RDML function. This means that the executing function starts (or spawns) another function to execute within the environment. The spawned function executes concurrently with, and completely independently of, the function that submitted it.

A shipped Visual LANSA system does not normally have IBM i job queue emulation enabled.

In this default environment the SUBMIT command works, but there is no inherent ability to queue the submitted jobs for deferred or serial execution.

If you do this:

```
begin_loop from(1) to(5)
    submit process(demo) function(test)
end_loop
```

then all five spawned functions will begin to execute as soon as they are submitted. This means that it is likely that all five spawned functions will end up executing immediately (and concurrently).

However, by using the IBM i job queue emulation facility you can queue up all five functions so that they are executed serially, or so that their execution is deferred until some predetermined time (e.g. overnight).

This type of job queuing is often called "batch processing".

Also see

[16.4.1 Establishing the X_JOBQ.DAT File](#)

[16.4.2 Starting, Stopping, Holding and Releasing Job Queues](#)

[16.4.3 Job Queue Priorities](#)

[16.4.4 Additional Job Queue Monitor Parameters](#)

[16.4.5 Submitting Jobs Across a Network](#)

[16.4.6 Implementation, Performance and Throughput](#)

16.4.7 Encrypting the Job Queue Details

16.4.1 Establishing the X_JOBQ.DAT File

To use this emulation facility you must first establish a file called X_JOBQ.DAT in the primary x_lansa directory (for example c:\program files\LANSA\x_win95\x_lansa under Windows 32-bit).

Note that when executing a 64-bit application, x_win64 replaces x_win95.

This file specifies, for all associated partitions, the names of the job queues and optionally, job descriptions which use this facility.

Any reference to job queues or job descriptions that are not defined in X_JOBQ.DAT remain unchanged and continue to execute in the default manner.

X_JOBQ.DAT is a text file that can be created and edited with most text editors. For example, X_JOBQ.DAT may be defined like this:

```
jobq=qbatch=c:\jobq\qbatch
jobd=qbatch=c:\jobq\qbatch
```

These X_JOBQ.DAT entries indicate that any SUBMIT command reference to job queue qbatch or job description qbatch should be routed, as a IBM i emulated job, into the directory c:\jobq\qbatch for execution at some later time by the "monitor" of that queue (more information about queue "monitors" follows).

Important things you must know about this file:

- Each line must be formatted,

<type>=<name>=<path> where:

<type> Must be "JOBQ" (job queue) or "JOBQ" (job description).

<name> Is the job queue or job description name which must conform to IBM i object naming conventions.

<path> Is the fully qualified path name to a directory that will be used as the job "queue". The directory must exist at the time you begin to execute any application that references a job queue or description in this file. The path name does not have to be associated with the queue or job description name in any way.

- Lines not formatted in this manner will be ignored. No error messages are issued for ignored lines.

Notes

- JOBQ= entries may exist by themselves.
- Every JOBQ= entry must have an associated JOBQ= entry that has exactly the same path details. This allows job description names to be associated with a job queue name.
- All JOBQ= names must be unique.
- All JOBQ= names must be unique.
- Every unique JOBQ= entry must have an associated unique directory path that must exist and be accessible at execution time. All users require write access to the directory. Job queue monitors require full read, write, update and delete rights to the directory.
- No individual entry line in X_JOBQ.DAT can exceed 256 bytes.
- Entries in X_JOBQ.DAT are all read into memory the first time that they are referenced. Subsequent references within the same x_run session refer to the details stored in memory. This means that changes to X_JOBQ.DAT may not be reflected in currently active x_run sessions/jobs.
- When you submit a job to a job queue in LANSAs on a windows environment, the job appears with a 'Q' extension, e.g., job T313330.Q50. The Q extension is given a unique alpha/numeric identifier for each job submitted in one second for a particular process. For example if job T313330.Q50 is submitted with other jobs in the same second, LANSAs will start assigning extensions Q51, Q52,..., Q5A, Q5B, Q5C, ... Q5Z. There is a limit of 36 jobs per second applicable for this unique identifier. To avoid any potential loss or corruption of submitted jobs, you should not submit more than 36 jobs per second. If your requirements have the potential to submit more than 36 jobs per second, you should include some logic to delay the submission of jobs.
- When jobs are submitted to an emulated IBM i job queue they always have their TPTH= parameter set to the same value as the path associated with the job queue regardless of how the TPTH= value of the submitting job is set.
- The Job Description and Job Queue Name XLANSAsJOBX has been reserved for LANSAs internal use. Do not define it in X_JOBQ.DAT

Once set up, entries in this file will cause all SUBMIT commands to examine its contents. If a match is found for the referenced job queue or job description name, then the submitted job will be "routed" to the specified directory.

The routing process consists of creating a series of binary files in the directory that contain details of the submitted job (e.g. request details, exchange values, LDA values, etc). These values are in binary format and they should not be

edited or changed.

The series of binary files representing the "queued" batch job will wait until their presence is detected, by the "monitor" assigned to the job queue.

16.4.2 Starting, Stopping, Holding and Releasing Job Queues

Once you have defined entries in X_JOBQ.DAT, and can submit jobs into the directory associated with a job queue, you need to understand how to start a "monitor" against a job queue.

To do this, simply use a normal x_run command like this:

```
X_RUN PROC=*STRJOBQ QNAM=QBATCH ... etc .....
```

This example would start a monitor running against the job queue defined in X_JOBQ.DAT with the name QBATCH.

Under Windows you should actually use:

```
START X_RUN PROC=*STRJOBQ QNAM=QBATCH ... etc .....
```

to start the monitor running as another process.

You can also control a job queue monitor by using:

```
X_RUN PROC=*HLDJOBQ QNAM=          to "hold" a job queue.  
<queue>
```

```
X_RUN PROC=*RLSJOBQ QNAM=<queue> to "release" a job queue.
```

```
X_RUN PROC=*ENDJOBQ QNAM=<queue> to "end" a job queue  
monitor.
```

To clear a job queue, simply erase all files in the nominated job queue directory (but only when the monitor is not active).

A monitor can only monitor a single queue.

Two or more monitors cannot monitor the same queue at the same time.

To see if a monitor is already attached to the queue, see if x_q_lck.sts exists in the associated directory. If it does exist, it indicates that a job queue monitor is attached, and the file should contain the process id (as text) of the current monitor. Also, the current status of the queue is stored in file x_q_sts.sts - either ACTIVE or HELD. If there is no monitor attached, the two *.sts files will not exist. This effectively means the job queue is stopped/ended.

16.4.3 Job Queue Priorities

Job queue monitors support the IBM i concept of job queue priorities.

Under IBM i, job queue priority is determined by the job description that the job is submitted under.

Each job on a job queue is assigned a priority in the range 0 to 9.

Jobs with the lowest priority value (i.e. closest to 0) execute first (i.e. they have the highest priority on the queue).

Within a single priority value, jobs are executed in order of arrival on the queue.

The job queue priority of the jobs that you submit can also be controlled by using the shipped Built-In Function `SET_SESSION_VALUE`.

The default priority is 5.

16.4.4 Additional Job Queue Monitor Parameters

When you use `x_run PROC=*STRJOBQ` to start the execution of a job queue monitor, you may also choose to alter these parameters:

`QCHK=nnnn` Specifies an integer value in the range 0 to 9999 indicating approximately how often (in seconds) an active monitor should wait before (re)checking the job queue directory for jobs to execute.

The default value is 10 seconds.

`QHLD=nnnn` Specifies an integer value in the range 0 to 9999 indicating approximately how often (in seconds) a held job queue monitor should wait before (re)checking the job queue directory for a release instruction.

The default value is 30 seconds.

`QENC=Y` Encrypt the job details before placing them on the queue. Refer to [16.4.7 Encrypting the Job Queue Details](#) for details.

16.4.5 Submitting Jobs Across a Network

If an X_JOBQ.DAT file was set up like this:

```
jobq=qbatch=c:\jobqs\qbatch  
jobq=qbnetw=s:\work\jobqs\qbatchn
```

where c: is the current PC's local hard drive and s: is a shared network drive, then it is easy to see how this facility can be extended to allow jobs to be submitted across a network.

If the current PC did a `START x_run PROC=*STRJOBQ QNAM=QBATCH` and the network file server PC did `START x_run PROC=*STRJOBQ QNAM=QBNETW`, then at any time a user of the local PC could elect to submit jobs locally (to queue QBATCH) or for execution on the network file server (to queue QBNETW).

Ideally the "submitting" PC and the "monitoring" PC should be using the same operating system and the same code page set. Failure to observe this guideline may lead to complexities in the exchanging of language-dependent character strings.

Where the submitting PC and monitoring PC may use different operating systems and/or code page sets, you should avoid designing applications that exchange (in any way at all) information that contains language-dependent character strings.

It is a requirement of this implementation that both the submitting and receiving PCs use identical system configurations in such things as partition definitions, supported languages, decimal points, date formats and so on.

Generally, in implementations like this, both the local PC and the network file server PC would have access to a common shared database as well.

16.4.6 Implementation, Performance and Throughput

The IBM i job queue emulation facilities were designed with these objectives in mind:

Persistence.

The ability to queue tasks when the "monitor" is not active. Submitted jobs persist beyond the duration of a session, and even when the entire machine is powered down and up again.

This is not supported when encryption of the job details is switched on.

This is because the encryption key is only valid whilst the Job Queue Monitor is running. Another instance of the Job Queue Monitor uses a new key and thus any existing jobs cannot be decrypted, and indeed are automatically deleted if any exist.

Portability.

The method used to emulate IBM i job queues has no deep operating system dependencies and can be easily ported to other multi-tasking operating systems in the future.

Volume capabilities and throughput rates similar to the IBM i facilities

The IBM i subsystem and job queue facilities are designed to handle relatively low batch job throughput rates of at most one job every 2 to 5 seconds.

Ultimately the throughput rate is governed by the amount of work that a submitted job does (i.e. in making jobs behind it on the queue wait), but rates beyond 1 job per 2 - 3 seconds, no matter how simple the submitted job, and no matter how powerful the processor, should not be expected.

IBM i application designers do not use the IBM i job queue capabilities to process jobs where throughput rates of 10's or 100's per second is required. When high rates like this are required, more advanced facilities such as data queues, named pipes, etc should be implemented via shipped or user defined Built-In Functions.

Summary

The actual throughput rate achieved by a job queue monitor, and even the rate at which jobs can be submitted to a queue depends upon many, many factors such as CPU speed, disk use, LAN traffic, CPU power, etc. Where very high throughput rates are an essential element of a design it is very strongly recommended that a prototype be constructed and verified early in the design

cycle.

16.4.7 Encrypting the Job Queue Details

To switch this feature on it's a simple matter of starting the Job Queue with the parameter QENC=Y. Once its started you submit jobs exactly the same way as when encryption is not being used.

Disclaimer

LANSAs does not warrant the effectiveness or otherwise of any of the cipher algorithms in the Open SSL library. You should perform your own due diligence before using any part of LANSAs which makes use of the ciphers. A suggested starting place is the book *Network Security with Open SSL* by John Viega et al, published by O'Reilly. The ciphers made available by LANSAs are as follows:

AES 256, Blowfish 128, CAST5 128, DES 64, DESX 192, Triple DES 2 Key, Triple DES 3 Key, IDEA, RC2(TM) and RC4(TM). There may also be patents current for some of these ciphers. It is up to you to ensure their usage does not contravene any patents. LANSAs accepts no responsibility whatsoever for any contravention of patents.

Technical Details

- Firstly, the only symmetric encryption algorithm currently supported is AES 256 using CBC mode (Cipher Block Chaining) and an Initialization Vector (IV) also known as a 'seed'. Other ciphers may be used but they are not currently supported by LANSAs and have not been tested. You use them at your own risk. There is an x_run parameter CIPH which is passed directly to the Open SSL API EVP_get_cipherbyname. Refer to the Open SSL documentation for further details.
- The public key must be RSA because it is the only OpenSSL public key algorithm that supports key transport. LANSAs uses 2048 bit modulus with RSA_F4 and Blinding ON (stops timing attacks).
- Envelope encryption is the usual method of using public key encryption on large amounts of data, this is because public key encryption is slow but symmetric encryption is fast. So symmetric encryption is used for bulk encryption and the small random symmetric key used is transferred using public key encryption.
- The Private Key is kept in the memory of the Job Queue Monitor. Thus there is no security issue except that you can debug the process, hence the right to debug should be revoked.

- The Public key, of course, is available to all users and is written out to a file in the job queue directory.
- A new key pair is produced every time the Job Queue is started.
- Each job has a new seed. This seed is essential as the data is very similar for each job queue file. The Open SSL crypto API default behaviour is to provide a random seed.
- Only the job queue monitor determines whether encryption is being used or not. If it is on, the key pair is created and the public key written out to a DER binary format file. Clients check for the existence of this public key file. If it exists, the client produces a random seed (IV) which is used to produce a unique symmetric key (Session Key) which is encrypted with the public key. The Session Key is then used to encrypt the job queue data. The IV, the encrypted Session Key and Encrypted job queue data, are written out to the job queue file. The structure of the encrypted job queue file follows:
 - Structure of the Job Queue file:
 <IV length><IV><Session Key length><Encrypted Session Key>
 <encrypted data length><encrypted data>
 - The RSA public key is named after the symmetric cipher lookup string, for example, aes-256-cbc.der. Thus the name of the file can be used to look up the symmetric cipher. Note that this may be confusing as this file contains the RSA asymmetric key, not the symmetric key.
 - When the Job Queue Monitor starts up, it firsts deletes all existing .der files and all existing job details before generating the Key Pair and outputting the new public key.

Responsibilities:

Job Queue Monitor

1. Generate key pair.
2. Output public key to <named symmetric cipher>.der e.g. aes-256-cbc.der.
6. Read the data file.
7. Decrypt using in-memory private key (RSA asymmetric cipher) and clients Session Key (named symmetric cipher).

Client

3. Read public key (RSA).
4. Generate Session Key for use with named symmetric cipher provided by server.
5. Create the data file, encrypting with appropriate keys.

- X_RUN Parameters:
 - CIPH - Cipher Name defaults to aes-256-cbc. case sensitive as it's passed as-is through to the OpenSSL library. This parameter is also available with the Built-In Functions GET_SESSION_VALUE and SET_SESSION.
 - QENC - Use encryption with Job Q Monitor. (This is ignored by submitters of jobs to Job Queue)
- If a job is submitted to an encrypted job queue, it can only be executed by that INSTANCE of the job queue Monitor. Once the Job Queue monitor is stopped and restarted, a new private key is generated which will not be able to decrypt the existing batch jobs. Clients do not need to be restarted, but neither can they submit any jobs until the Job Queue Monitor is running and has generated the Public Key.

@@@ here

16.5 The RUNSQL Utility

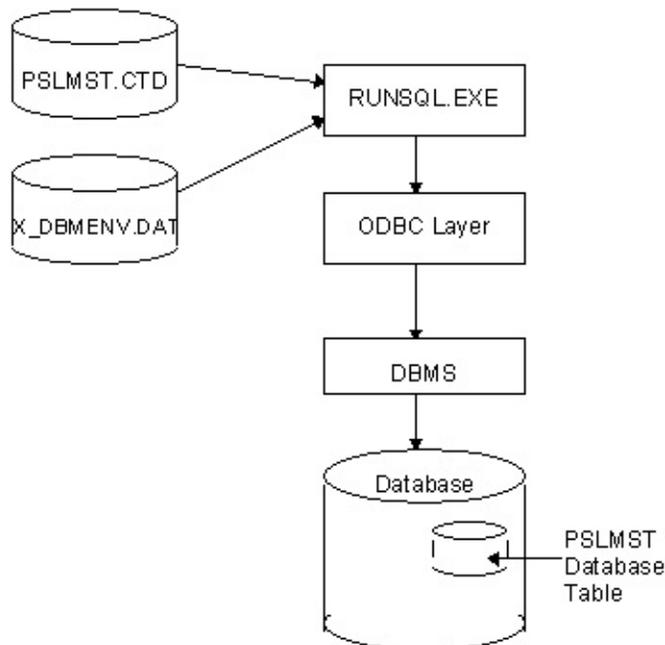
16.5.1 Configuration Notes - Creating Tables and Indexes

All Visual LANSA systems are shipped with a utility named RUNSQL.

RUNSQL can be used to automatically create the definition of a table into any supported DBMS system.

RUNSQL, combined with a .CTD (Common Table Definition File) file created by Visual LANSA during table compilation, form the essential ingredients that you need to move table definitions (not data) between different supported DBMS systems.

To understand how RUNSQL works consider this diagram:



If you imagine that you are attempting to transfer the definition of a table named PSLMST (that you have previously defined and compiled in your development environment) into another DBMS, then the key things shown in this diagram are:

- When the RUNSQL utility is invoked it reads in the file named PSLMST.CTD. This is the "Common Table Definition" (CTD) of table PSLMST that is created by Visual LANSA whenever you compile a table in your development environment. It defines table PSLMST and its associated views and indices in a common cross platform / cross DBMS format (full details of the format of .CTD files can be found in another section of this guide).

- RUNSQL also reads in a standard Visual LANSA file named X_DBMENV.DAT (Database Environment Definitions) that defines the unique characteristics of the DBMS that it is about to work with.
- By using PSLMST.CTD and X_DBMENV.DAT the RUNSQL utility can assemble the unique "create" commands appropriate for the selected DBMS.
- Once the "create" commands are assembled the DBMS is invoked (via ODBC in Windows environments) and it is asked to create the necessary table, view, indices, etc.

RUNSQL is a simple program. It has the following positional and non-positional parameters:

1	The (qualified) name of the .ctd (Common Table Definition) file that contains the definition of the table to be created. Common Table Definition files are created whenever you create a table in your Windows development environment. The.ctd files can be found in the X_LANSA\X_ppp\SOURCE directory (where "ppp" is the partition identifier).
2	The name of the database or data source that the table is to be created into. Typically this parameter is passed as LX_LANSA.
3	Commitment Option. Must be Y or N and indicates whether a commit operation is to be issued after the table has been successfully created. You should always set this parameter to Y.
4	Reporting Option. Must be Y, N or F to indicate the level of reporting that RUNSQL should use. Y = Report on all messages and warnings. N = Do not report any messages or warnings. F = Report on fatal messages only.
5	The type of database. This value is used to locate the database characteristics in the specified "X_DBMENV.DAT" file. Some of the standard shipped database types are: - SQLANYWHERE (Sybase Adaptive Server Anywhere and Sybase SQL/Anywhere) - MSSQL (Microsoft SQL/Server)
6	The user profile / password to be used when attempting to connect to the specified database or data source.

	For example SA/TEST specifies that user profile SA with password TEST be used when connecting to the database or data source.
7	Specifies the directory in which the "X_DBMENV.DAT" file can be found.
8	Specifies the collection. Default is specified in the .ctd file
9	CTD Connection data option. Must be Y or N and indicates whether to use the connection information contained in the .ctd file. Only PC Other Files will have connection data in the .ctd file
10	Prompt User ID/ Password option. Must be Y or N and indicates whether to use the User ID and Password in the .ctd file (N) or to prompt for a new pair of values (Y). This is ignored unless CTD connection data is being used.

Non-Positional Parameters

OLDCTD=	Old .ctd file name. This is the .ctd file that was last used to create/change the table. The new and the old CTD are compared and any changes or new columns are added to the table without deleting the existing data.
---------	---

Note that non-positional parameters can be placed anywhere on the command line separated by spaces from the other arguments.

For example, this command executed from the x_Lansa\source directory compares myfile.ctd to myfile_old.ctd and makes the changes to the table. Note that it also uses the x_dbmenv.dat file from the parent directory - – which in this case is the x_lansa directory:

```
..\execute\runsql myfile.ctd OLDCTD=myfile_old.ctd LX_LANSA Y Y
SQLANYWHERE DBA/SQL
```

16.5.1 Configuration Notes - Creating Tables and Indexes

You may also configure RUNSQL using the environment variables X_RUNSQL_CREATE_TABLE and X_RUNSQL_CREATE_INDEX. These variables allow DBMS-specific strings to be appended to the "create" commands before RUNSQL executes them.

Note: No syntax checking will be done. If the appended information is bad syntax, the DBMS call will fail and an error code will be returned.

The following example shows how to use this feature to separate tables and indexes into different storage areas.

For SQLANYWHERE on Windows:

```
SET X_RUNSQL_CREATE_TABLE="IN DATA"  
SET X_RUNSQL_CREATE_INDEX="IN IDX"
```

(DATA and IDX are SQL/Anywhere dbspaces created earlier with the CREATE DBSPACE command.)

For the example table PSLMST, RUNSQL will create the table PSLMST with the following statement:

```
create table PSLMST (...column list...) IN DATA
```

and the relative record number index with the following statement:

```
create unique index PSLMST_R on PSLMST (x_rrno) IN IDX
```

For ORACLE on LINUX:

```
X_RUNSQL_CREATE_TABLE="tablespace DATA"; export  
X_RUNSQL_CREATE_TABLE  
X_RUNSQL_CREATE_INDEX="tablespace IDX"; export  
X_RUNSQL_CREATE_INDEX
```

(DATA and IDX are Oracle tablespaces created earlier with the CREATE TABLESPACE command.)

For the example table PSLMST, RUNSQL will create the table PSLMST with the following statement:

```
create table PSLMST (...column list...) tablespace DATA
```

and the relative record number index with the following statement:

```
create unique index PSLMST_R on PSLMST (x_rrno) tablespace IDX
```

Note: Primary Key Indexes will only use the X_RUNSQL_CREATE_INDEX value if the DBMS entry in x_dbmenv.dat contains the following settings:

SUPPORTS_PRIMARY_KEY=NO
CONVERT_PRIMARY_KEY_TO_INDEX=YES

16.6 Font Considerations

The font determines the characteristics of letters, numbers, and symbols displayed on your PC screen. Fonts may have different sizes (width, height, etc.), attributes (bold, underline, etc.) and styles (Courier, System VIO, Helvetica, Swiss, etc.).

Some fonts are described as being proportional fonts while other fonts are described as fixed or non-proportional fonts. A proportional font uses a different amount of space to display different characters. A fixed font uses the same amount of space to display each character.

For example, the characters "EEE" would require more space than "iii" if a proportional font were used. If a fixed font were used, the same space is used by both "EEE" and "iii".

The advantage of using fixed fonts is that the size of the input field displayed will indicate the exact number of characters which can be entered. If a fixed font is used, a three character input field will appear to display space for exactly three characters, remembering that "EEE" and "iii" use the same space.

If you have selected a proportional font, the amount of space used in the input field will depend on the text entered. You may only see "EE" even though "EEE" was entered. This truncation is a result of the fonts selected and not the application. Be careful when selecting proportional fonts.

If you are retrieving the cursor location (e.g. CURSOR_LOC PARAMETER for DISPLAY, REQUEST or POP-UP), you should use fixed (i.e. non-proportional) fonts.

The X_UIM (User Interface Manager) allows you to change the font being used to present information. The actual font selection facility is a supplied part of the Windows operating environment.

Although any font may be chosen, the choice will depend upon the resolution of the monitor being used. The following fonts are recommended as being the most suitable for the most common VGA monitor resolutions:

MS Sans Serif Size 8

MS Serif Size 8

Arial Size 8

Verdana Size 8

System Size 10

Note that changing fonts is not a common operation. Once a font has been chosen it is usually used from then onwards with no change. When you change fonts the following may happen:

- The current screen may appear strangely with information out of place. This happens because the font change does not trigger a resizing and redrawing of the current screen. Exit from the application and then restart it in the new font.
- If you change fonts and do not exit from and restart the application, the next screen chosen may cause the entire window to disappear and then re-appear in the new size/font. This happens when the UIM decides it has to resize the entire presentation window it is using. To avoid this problem, exit from the application and then restart it after changing fonts.

Finally, you should know that the details of the font that you select are stored and remembered from session to session.

They are stored on, and associated with, the workstation that you are using, not with any user profile that you are using.

If you change the font used on a workstation you are changing it for all users of the workstation. If you move to another workstation your selected font will not follow you to the new workstation.

16.7 Sizing RDML Windows

All windows presented by DISPLAY and REQUEST RDML commands are sizable. This means that you can stretch or shrink the window into any size or shape that you desire.

The stretching and shrinking process is performed in the usual manner by positioning the cursor on the window border (until a double headed arrow appears) and then holding down the left mouse button while moving the window border in or out as desired.

When using this facility you should be aware of the following:

- The window shape that you select will be remembered and used for all following DISPLAY and REQUEST commands.
- Pop-up windows resulting from RDML POP_UP commands are sized and positioned relative to the shape of your basic full window.
- Text and field information is presented in all windows relative to your basic full window. If your basic window is too narrow your entry fields may be too short and textual information may be truncated. In this case, stretch your basic window so that it is wider. If your basic window is too short, then font details may not be shown or may overlap other fields. In this case stretch your basic window so that it is longer.
- Generally the size of the window that you select must be related to the font that you are using. If you are using a large font (size 10 or larger) then you must use larger windows. If you are using a small font (size 8 or smaller) then you can use smaller windows.

16.8 Windows 64-bit Support

This page is

It is recommended that you only enable 64-bit support if installing a Build machine and it is mandatory that the application is required to be 64-bit. For example, it's a required corporate standard. Some drawbacks of enabling it are:

- Compiles take twice as long as both 32-bit and 64-bit DLLs are always built.
- Functions which use DISPLAY, REQUEST or POPUP commands will fail to compile; even 32-bit DLLs.
- You must obtain your own 64-bit compiler.

LANSA Features that do not function or have no support

- Graphics Server is not supported. There are many modern alternatives that will work on both 32 bit and 64 bit. Use them. This maps to the primitive PRIM_GRP.
- Explorer Component AutoRefresh Property does not function.
- ZIP BIFs are not supported.
- DISPLAY, REQUEST and POPUP commands are not supported in 64-bit applications.
- Web Functions are not supported.
- Specialised LANSA Built In Functions (BIFs) are not supported in 64-bit. This is because the development environment is 32-bit and thus they are not compatible.

Installation Considerations

- It is presumed that a developer does NOT enable 64-bit support. The intention is that it's only the Build Machine that has 64-bit support enabled. So it is not possible to ONLY have 64-bit support. Some noticeable consequences of this mean that when 64-bit support is enabled:
 - Both the 32-bit and 64-bit compiles are performed, which takes longer and thus may not be appropriate on a developer's machine.
 - Both 32-bit and 64-bit Windows Installer MSI packages are built.
- Visual Studio 2010 Professional (or later) or Visual Studio 2012 Express for Desktop (or later) are required to support 64-bit compiling. The compiler shipped with the Visual LANSA install does not support 64-bit compiles.
 - If Visual Studio is installed before LANSA then it will be detected and

will be the compiler used by LANSA.

- If a supported compiler is not installed, the LANSA-shipped compiler will be installed and enabled. To enable 64-bit compiling, install one of the compilers that supports 64-bit compiling and then change this registry entry to disable the LANSA-shipped compiler:
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\LANSA\M to 0. On a 32-bit PC it is:
HKEY_LOCAL_MACHINE\SOFTWARE\LANSA\MicrosoftCompile
- If the latest version of Visual Studio installed is not one that supports 64-bit compiling, just install one that does. LANSA will detect it when it is next started.
- If you compile a Function which contains DISPLAY, REQUEST and POPUP commands (which are not supported) it will fail to compile at all - even the 32-bit compile. This is another reason why it is better not to enable 64-bit support on Developer's machines. If a Developer needs to work on both RDML Functions and 64-bit applications then 2 systems may be installed on their machine which use the same repository.

Programming Considerations

- There are no increases in the maximum size of any LANSA feature. For example, the maximum size of an RDMLX List is still 2 billion rows, with each entry being 2 billion bytes long. That is, existing limits are considered sufficient. This also means there is greater compatibility between 32-bit and 64-bit applications. For example, the Built In Functions SND_TO_DATA_QUEUE and RCV_FROM_DATA_QUEUE may be used interchangeably. Job Queue Emulation can use either a 32-bit or 64-bit Job Queue Monitor and jobs may be submitted from either 32-bit or 64-bit. Note that the 64-bit Job Queue Monitor will execute the submitted job as 64-bit, no matter which platform submitted the job.
- PC Other Files which are **loaded** using a 32-bit ODBC driver will need to create a 64-bit DSN with the same name as that used to load the file or use CONNECT_SERVER when **deployed** to re-direct IO to a 64-bit driver.
- To put an ActiveX in LANSA RDML there must be a registered 32-bit version of the ActiveX. To EXECUTE the ActiveX, a version must be registered which is of the same processor architecture as the LANSA runtime. That is, if the LANSA runtime is 64-bit then the 64-bit ActiveX must be registered on the deployed PC.

32-bit and 64-bit applications accessing the same database

Considerations when both a 32-bit application and a 64-bit application are accessing the same database, particularly when deploying an application into a production system:

- LANSAs recommends using either 32-bit or 64-bit applications. It makes it far simpler. For example, when using SuperServer, only use a 64-bit server if you use both 32-bit and 64-bit clients. As the clients are not directly accessing the database, there is no complication. It is better to choose to exclusively use one or the other.
- Assign relative record numbers using auto-generation. If relative record numbers are assigned using external files, duplicates will occur unless the RPTH parameter is assigned to the same path for both 32-bit and 64-bit applications. A file that is currently using external files may be changed to auto-generation using the Upgrade tool feature Convert Files to Use Identity Column.
- Table upgrades are identified by comparing the previous CTD file to the new CTD file being installed. Thus only the first system upgraded should upgrade the database. This is why database upgrade defaults to off during an MSI install and why per-user installs disable database upgrade. If an existing OAM is not there for 64-bit but is for 32-bit, and vice versa - which is the latest OAM? This needs to be controlled by the user. If 32-bit is the first environment to be installed, continue that way for all Upgrades and Patches. Once the 64-bit environment is at the same level, there is the choice that the Upgrade/Patch database change machine can be switched, but it is inadvisable. Be consistent and use one machine from the beginning.

Notable Environmental Differences

- The system directory for 32-bit applications is of the form x_win95\x_lansa. For 64-bit applications it is x_win64\x_lansa. Therefore system variables like *SYS_DIR return a different value.
- Visual LANSAs is a 32-bit application. Hence interaction between Visual Lansas and 64-bit generated DLLs cannot occur.
- 32-bit OAMs are always built and will always be built as Visual LANSAs requires the 32-bit OAM to unload and load the data from the table. The 64-bit build command always skips the SQL table build, presuming that 32-bit has already done it.
- Windows Installer has a known defect which converts the Target directory in a Shortcut from c:**program** files to c:**program files (x86)**. Nonetheless the shortcut still works correctly as if it was c:**program** files. Even if the 32 bit

version of the Application is installed in **c:\program files (x86)**, it does not get executed, it is still the 64-bit version. See this web link [32bit MSI on 64bit OS: Converting shortcut target path of 64bit app to 32 bit Path](#).

- A similar situation occurs with Windows\system 32. The shortcut looks OK but it does not find the object. It is not valid to create a shortcut that points to this directory.

16.9 Linux Differences

- Linux is a case-sensitive operating system.

By convention, directories, files, printer names and so on are usually in lower case. The exception to this is the .RRN files which are still in upper case.

- Instead of the directory "execute", LANSAs on Linux uses the directory "bin".
- Instead of a back slash (\) to separate directories in a path, Linux uses a forward slash (/).

When looking for files or executing commands under Linux, change all upper case to lower case and use forward slashes (/) and not back slashes (\).

For example, if the path to the X_RUN executable under Windows was

\X_LANSA\EXECUTE\X_RUN

it would become, under Linux:

/X_LANSA/bin/X_RUN

Refer to the *Deploying LANSAs Applications on Linux Guide* for more information.

16.10 Code Page Considerations

If characters are not properly translated when loaded from the IBM i and into Visual LANSA, you need to review the code page definitions on the IBM i.

Code pages are sets of definitions for each character on your keyboard. It is possible that the characters used on the IBM i (5250 keyboards) are not properly mapped to characters used on the PC.

If characters are not properly translated when executing Visual LANSA generated applications, you need to review the conversion tables and files defined to Visual LANSA. It is possible that the characters used for one operating system may be different for another operating system.

Characters which are commonly wrong include # or @.

It is very important that you determine the correct character set conversions right from the start. Spend the time considering your requirements before beginning to build any applications, otherwise you will need to re-build your applications each time you change the conversion tables.

There are three places that you should review for character set conversion:

1. The x_defppp.h header file (where ppp = partition identifier) in each partition. The contents of this file are included in [16.11 Regional Settings](#).

Each time you change the x_defppp.h header file you will have to re-build all your "entry point" processes (i.e. the ones through which you enter your applications).

2. LANSA SuperServer conversion tables defined in Built-In Function DEFINE_OS_400_SERVER.
3. Translation tables specified using PCMAINT.

16.11 Regional Settings

After installation, each ...\`X_LANSA\X_ppp\SOURCE` directory should contain a file called `x_defppp.h` (where `ppp` is the name of the partition). For example, if the partition is `SYS`, then the file will be called `x_defsys.h`.

The `x_defppp.h` file defines execution settings for your functions that are unique to the partition. (It is similar in concept to the `DC@A01` data area on the IBM i). These settings are globally defined to all PROCESSES (and subsequently the functions) in a specific partition.

For example, your decimal point character may be `,` rather than `.` Using this file, you can specify the character that you require once, for all processes and functions in the partition.

The file is formatted as a C header file. It is very easy to understand and to change with any source/text editor, if necessary. You will find the relevant options and values documented within the file.

Any changes made to the values contained in the `X_DEFPPP.H` file, will require (re)compilation of ALL ENTRY POINT PROCESSES before the changes take effect. Once ALL entry point processes have been (re)compiled you will need to exit and re-enter your LANSA application for the new values to be picked up at execution time.

You should also note that the following values can be set from the LANSA `X_RUN` command or as a system environment variable:

```
X_AUTOMATIC_HELP,X_CENTURY_COMPARE_DATE,X_CENTURY_C
X_CENTURY_LESS_DATE,X_DATE_SEPARATOR,X_CURRENCY_SYM
X_DECIMAL_POINT_CHAR,X_TIME_SEPARATOR,X_DOLLAR_SIGN_
X_HASH_SIGN_CHAR,X_AT_SIGN_CHAR,X_GEN_AT_SIGN_CHAR,
X_STANDARD_MESSAGE_FILE
```

The order of precedence for the setting of these values is:

- A. Values specified in `XENV=` parameters on the `X_RUN` command.
- B. Values specified in normal system environment variables.
- C. Multi-national Values specified in the Development Environment
- D. Country-specific information read from the Windows settings. These values are always available, so in order that the values in `x_defxxx.h` can still be used, the use of these values can be switched off. To do this define the registry value "OSRegionalSettings" in

HKEY_LOCAL_MACHINE\Software\LANSA\X_LANSA registry key as a DWORD and set its value to 0.

E. Values specified in this file.

For example, say that you needed to set the X_CENTURY_LESS_DATE value to "20". You can do this using three different methods. These methods can be used independently or together:

- Specify the value in this file by changing the line below to #define X_CENTURY_LESS_DATE "20" and then recompiling all entry point processes in your application, This will effectively lock your application into this value. However, this value will not normally be used by application pieces running in SuperServer mode or as services to LANSA Open applications because they do not have the concept of an entry point process available to them. In such cases you should use method B or C as well to achieve the required results in all situations.
- Set an environment variable for the environment in which application is executing. By putting the operating system command SET X_CENTURY_LESS_DATE=20 into your operating system start up you can cause the appropriate value to be set. A value set this way overrides any value specified by method A. When setting a value this way you should take care to ensure that the value is appropriate and valid, as no form of validation is applied to the value specified.
- Set a LANSA environment variable by putting the value into the X_RUN command. For example, X_RUN PROC=TEST LANG=ENG XENV=X_CENTURY_LESS_DATE=20 XENV=X_AUTOMATIC_HELP=Y will set the century compare date and automatic help option to the values specified, overriding anything specified by environment variables (method B) or in this file (method A). Note that a value set this way overrides any value set by methods 1 or 2. When setting a value this way you should take care to ensure that value is appropriate and valid, as no form of validation is applied to the value you specify.

There is a fourth method of obtaining the multinational values:

X_DOLLAR_SIGN_CHAR, X_HASH_SIGN_CHAR and X_AT_SIGN_CHAR.

These values are obtained from the LANSAPC registry entry. This is primarily used for LANSA objects that are used to extend the Development Environment such as the Deployment Tool.

Note: If running in LANSa SuperServer, ensure that the Server setting matches the Client setting for X_DECIMAL_POINT_CHAR to avoid values exchanged from a function called via the Built-In Function CALL_SERVER_FUNCTION losing their decimal places.

17. Execution Control

- [17.1 Using X_START as a Front End to X_RUN](#)
- [17.2 The X_RUN Command](#)
- [17.3 X_RUN Parameter Summary](#)
- [17.4 X_RUN Parameter Details](#)
- [17.5 Permanently Specify X_RUN Parameters](#)
- [17.6 Database Connections](#)
- [17.7 The .XQ* Files](#)
- [17.8 Lock Timeout](#)
- [17.9 User Instructions for Microsoft Exception or Dr Watson](#)

17.1 Using X_START as a Front End to X_RUN

The X_START facility is shipped with all Visual LANSA systems.

Using X_RUN.EXE to invoke Visual LANSA facilities directly from an icon on your desktop often means that you may end up with many individual icons and when a command parameter needs to be changed it needs to be changed many times.

This problem can be overcome by using the X_START utility. For example, imagine you had the following X_RUN command associated with an icon on your desktop:

```
X_RUN PROC=TEST01 PART=DEM USER=QPGMR
```

then by altering format of the X_RUN command behind the icon to:

```
X_START X_RUN PROC=[TEST01/Process Name/PROCESS]  
PART=[DEM/Partition Identifier /PARTITION]  
USER=QPGMR .... etc .....
```

you can:

- cause the values of PROC= and PART= to be prompted.
- cause the default values of PROC= and PART= to be remembered and for their "last used" values to be exchanged between the X_START commands behind different icons.

The way that X_START works is very simple. It processes the entire command line looking for [//] or { // } formatted prompt requests.

A [//] or { // } formatted prompt request must always be formatted exactly like this:

```
[default value / description / symbolic name]
```

or this:

```
{default value / description / symbolic name}
```

Generally you should use the [//] notation in Windows environments.

For example, this is the command invoked when you select Execute Process on a Workstation from the LANSA menu:

```
%tit%Execute process on workstation  
%hlp%x_start.009  
%basepath%\x_lansa\execute\X_RUN.exe  
PROC=[Name/Process/PROC]  
LANG=[ENG/Language/LANG]
```

PART=[DEM/Partition Identifier/PART]
USER=[QPGMR/LANSA User/USER]
%WIN%DBUS=[DBA/Database User/DBUS]
%WIN%PSWD=[*password/Database Password/PSWD]
DBII=[LX_LANSA/Database Name/DBII%reg%LX_DBName]
%W95%DBUT=
[SQLANYWHERE/Database Type/DBUT%reg%LX_DBType]
%WNT%DBUT=
[MSSQLS/Database Type/DBUT%reg%LX_DBType]
%W95%CMTH=[E/Communication Method/CMTH]
%WNT%CMTH=[C/Communication Method/CMTH]
%W95%CDLL=
[E32APPC.DLL/Communications DLL to Use/CDLL%reg%LX_CommsDI]
%WNT%CDLL=
[WCPIC32.DLL/Communications DLL to Use/CDLL%reg%LX_CommsDI]
PRTR=[LPT1/Default Printer/PRTR]

This function causes the *Execute process on workstation* dialog box to appear.

Field	Value
Process	PCXDEMO1
Language	ENG
Partition Identifier	DEM
LANSA User	QPGMR
Database User	DBA
Database Password	
Database Name	LX_LANSA
Database Type	SQLANYWHERE
Default Printer	LPT1
Debug	N
Trace	N
Max Trace Lines	20000
Trace level	4
Trace Categories	ALL
Heap Validation	X

The values entered are then substituted into the command. If XXXXXX was keyed as the process name and SYS as the partition identifier, then the command that is assembled for execution would be:

X_RUN PROC=XXXXXX PART=SYS USER=QPGMR etc

Additionally, the value XXXXXX would be remembered with the symbolic

name PROCESS and the value SYS would be remembered with the symbolic name PARTITION. This means that the next time the X_START facility is invoked and a [//] or {//} prompt has the symbolic name PROCESS then the remembered value XXXXXX will be used in preference to any default value specified in the [//] or {//} prompt.

The symbolic names and their values are remembered in a simple text file named X_START.SAV that is created and updated into the current directory of the process executing the X_START request. If you suddenly lose your last set of values (i.e. they revert to their defaults), then the most likely reason is that you have altered the current directory of your application such, that the X_START.SAV can no longer be found.

The data stored in file X_START.SAV is logically formatted as <symbolic name><value><symbolic name><value> and up to 1024 symbolic names may be in use at any time. Neither the symbolic name nor its value should ever be more than 256 characters in length.

Also see

[17.1.1 Rules, Limitations and Guidelines](#)

[17.1.2 Commands and Special Variables](#)

17.1.1 Rules, Limitations and Guidelines

The following rules, limitations and guidelines apply to the use of the X_START facility:

- No default value, remembered value, description or symbolic name can be more than 256 characters long.
- All [//] prompt requests must be precisely formatted [default/description/symbolic name] using the '/' character to delimit the areas between the [] characters.]
- All { // } prompt requests must be precisely formatted {default/description/symbolic name} using the '/' character to delimit the areas between the { } characters.
- Special values *NONE and *PASSWORD may be used in the default and/or symbolic name section of any [//] or {//} prompt area. *NONE indicates that no value exists and/or that the prompt value should not be saved in file X_START.SAV. *PASSWORD indicates that the prompt is for a password field (i.e. that entry made into the prompt should not be readable) and that the value should not be stored in file X_START.SAV.
- The correct format for a password field prompt is [*PASSWORD/description/symbolic name] or {*PASSWORD/description/symbolic name}. You cannot cause a password field to adopt a default value.
- No more than 1024 symbolic names (and their associated values) can be stored in any X_START.SAV file.
- X_START looks for (and saves) the file X_START.SAV into the current directory of the process invoking the X_START facility.
- The X_START facility can extract the command line definition to be processed either directly from the icon command line or from a file.

To extract the command definition from a file, simply use the name of the file containing the command line definition prefixed by an '=' (equal) sign or an '@' (at) sign as a single argument to X_START. For example:

```
X_START %basepath%\x_lansa\execute\X_RUN  
proc=[X/Process/PROCESS]
```

and

X_START =TEST.DTA

are functionally identical operations when TEST.DTA is a file that contains these 2 lines:

```
%basepath%\x_lansa\execute\X_RUN  
proc=[X/Process/PROCESS]
```

- The default characters that denote the start of a prompt are '[' and '{'. These may be altered by inserting the following into the current X_START.SAV file:

```
X_START_OPEN_BRACE1  
y
```

or

```
X_START_OPEN_BRACE2  
y
```

where y is the character you wish to use in place of the '[' or '{' characters. You should only alter the default value in extreme situations involving code page conflicts, etc.

- The default characters that denote the end of a prompt are ']' and '}'. These may be altered by inserting the following into the current X_START.SAV file:

```
X_START_CLOSE_BRACE1  
y
```

or

```
X_START_CLOSE_BRACE2  
y
```

where y is the character you wish to use in place of the ']' or '}' characters. You should only alter the default value in extreme situations involving code page conflicts, etc.

- The character that separates strings within a prompt is the forward backslash '/'. This may be altered by inserting the following 2 lines into the current X_START.SAV file:

```
X_START_SEPARATOR  
y
```

where y is the character you wish to use in place of the '/' character. You should only alter the default value in extreme situations involving code page conflicts, etc.

- The string "OK" appears by default on the OK button of the prompt dialogue. You can change this string by inserting the following 2 lines into the current X_START.SAV file:

```
X_START_OK  
yyyy
```

where yyyy is the string that is to appear on the OK button. String yyyy should of course be sensibly sized.

- The string "Cancel" appears by default on the Cancel button of the prompt dialogue. You can change this string by inserting the following 2 lines into the current X_START.SAV file:

```
X_START_CANCEL  
yyyy
```

where yyyy is the string that is to appear on the Cancel button. String yyyy should of course be sensibly sized.

- The string "Parameter Help" appears by default on the Parameter Help button of the prompt dialogue. You can change this string by inserting the following 2 lines into the current X_START.SAV file:

```
X_START_PARMHELP  
yyyy
```

where yyyy is the string that is to appear on the Parameter Help button. String yyyy should of course be sensibly sized.

- The string "General Help" appears by default on the General Help button of the prompt dialogue. You can change this string by inserting the following 2 lines into the current X_START.SAV file:

```
X_START_GENLHELP  
yyyy
```

where yyyy is the string that is to appear on the General Help button. String yyyy should of course be sensibly sized.

- The previous points mean that the symbolic names:

```
X_START_OPEN_BRACE1, X_START_OPEN_BRACE2,  
X_START_CLOSE_BRACE1, X_START_CLOSE_BRACE2,  
X_START_SEPARATOR, X_START_OK, X_START_PARMHELP,  
X_START_GENLHELP and X_START_CANCEL are reserved and  
should not be used in [//] or {//} prompts as symbolic names.
```

- Assembled commands are always assumed to be being used to invoked .EXE programs, so the string .EXE will be automatically added to final commands as appropriate.
- The X_START.EXE program should only ever reside in the \X_LANSA\EXECUTE\ directory of the current LANSA system. Do not place it in any other directory.
- A file named X_START.SDH may also optionally reside in the \X_LANSA\EXECUTE directory. This file is used to support parameter and general help text and is formatted:
HELP=XXXXXXXXXX
 <lines of text>
HELP=XXXXXXXXXX
 <lines of text>
 where **XXXXXXXX** is the symbolic name of the parameter to which the help text applies or the general help identifier of the prompt.
- For full working examples of prompted X_RUN commands you should refer to files X_START.001 -> X_START.010 in your \X_LANSA\SOURCE directory and to file X_START.SDH in your \X_LANSA\EXECUTE directory.
- There is an optional merge file, which provides the ability for another program to dynamically provide values for symbolic names. Each line in the merge file has the format:

%<variable name>%=<value>
e.g. %proc%=MYPROC

The variable name must be 4 characters long followed immediately by an '='. The value is all the characters from the '=' to the end of the line. The variable name must not be a pre-defined variable name. This results in a fatal error at execution time.

17.1.2 Commands and Special Variables

The following commands and special variables may be used anywhere in a command string except inside a [//] or {//} prompt. Examples of the use of all these commands and special variables can be found in the XST files in \X_WIN95\X_LANSA\SOURCE shipped with Visual LANSA to provide custom execute dialogs from within the LANSA development environment:

Variable Name	Description
%basepath%	It is substituted with the name of the path up to (but not including) the X_LANSA directory of the current Visual LANSA system. For example c:\x_win95.
%browser%	Determine the user's browser and expand to the full path of the browser
%editor%	It is substituted with the string NOTEPAD.EXE under Windows.
%if% & %endif%	<p><code>%if%<expression>?<true text>:<false text>%endif%</code></p> <p>Include the 'true text' if the expression is true, otherwise include the 'false text', if it exists. The expression and text cannot include a prompt. The expression can either be an "=" or "!=" type of expression. The comparisons are case insensitive. The if statement finishes at the %endif%. The if statement cannot be nested.</p> <p>E.g. The following line compares the variable LANG to the string 'NAT' and if it is not equal, puts a '+' followed by the value of %lang% in the command line:</p>

```
%if%%lang%!=NAT?+%lang%%endif%
```

The following 4 lines prompt for "Debug", "Device Name" and "Message Queue", and if %dbug% is equal to 'Y', puts "+BDEBUG+%dvic%+%msgq%" on the command line:

```
%promptonly%[N/Debug/DEBUG]
%promptonly%[DSP01/Device Name/DVIC]
%promptonly% [DSP01/Message Queue/MSGQ]
%if%%debug%=Y?
+BDEBUG+%dvc%+%msgq%%endif%
```

%JavaClient%	The Java-Client physical path.
%noshowdialog%	Do not display the dialog. Use the defaults and merge values as if the user had just pressed OK on the dialog without changing any values. This is provided for use in the merge file so that a calling program can control whether or not to display the dialog.
%nospaces%	Do not add a space for each new line. Spaces must be manually included where necessary. This is primarily provided in order to assemble a URL from separate lines in the X_START file
%promptonly%	Only prompt for the value that follows. Do not insert the text at that point in the file. The dialog will just set the value of the symbolic name. A reference to the Symbolic Name ANYWHERE in the X_START file will insert its value there. See %if % examples
%show%	Causes the final command (after prompting) to be displayed before an attempt is made to execute it. This is useful for debugging custom scripts.
%<symbol>%	All symbolic names provided in prompts are accessible via this syntax. This includes values that are prompted for and the symbolic names defined in the merge file. The maximum number of symbolic names is 1024. If there are more symbolic names than this they are ignored
%workpath%	Sets the current directory to this value when the assembled command line is executed, e.g. when X_RUN.exe is executed. If this is not specified, the current directory is left unchanged.

Note:

When x_start is called from the Visual LANSA Execute Dialogs the current directory is set to \x_win95\x_lansa\source, which is the same as the location of the XST files

- %<environment variable>% If a match is not found for the variable name, then it is evaluated as an environment variable. If it exists, its value is substituted for the variable name.
- %X_RUN_specific% If %basepath% of X_RUN.exe is detected then the value of the following %X_RUN_specific% is evaluated.
If %basepath% of X_RUN.exe is not detected then any %X_RUN_specific% is ignored.
- *password A special value for a default value. Indicates that the parameter is a password and characters entered are masked with asterisks. A value must be entered
- *password_optional The same as *password but a value is optional
- *optional_data_<value> A special value for a default value. Indicates that entering a value is optional. If there is text for <value> it's the default value. E.g. *optional_data_ will display an empty field. *optional_data_adefault will display 'adefault' in the field

Note: If any of these commands are mistyped and the '%' is the first character on the line, the command will be taken as a Symbol Name and the rest of the line will be set as the value of that symbol. The resulting command line will look as if the whole line has been ignored.

17.2 The X_RUN Command

The X_RUN command is used to execute an application from a command line or program icon. It initiates the LANSAX execution environment and executes the specified application.

X_RUN has a number of compulsory and optional parameters which may vary depending on the platform on which it is used.

For example, this X_RUN command is used to bring up the menu MYPROC:

```
X_RUN PROC=MYPROC PART=DEM LANG=ENG  
USER=USERID DBID=LX_LANSA
```

Under Linux, this X_RUN command is used to print a report on printer lp0 using the MYREPT report function IN MYPROC process:

```
X_RUN PROC=MYPROC FUNC=MYREPT PART=DEM LANG=EN  
USER=USERID DBID=LANSA PRTR=lp0
```

The X_RUN command can be found in

- the x_lansa\execute directory under Windows
- the \$LANSAXROOT/x_lansa/bin directory under Linux.

The X_RUN command has standard parameters as well as parameters for more advanced use. These parameters are described in [17.3 X_RUN Parameter Summary](#) and [17.4 X_RUN Parameter Details](#).

X_RUN parameters and their arguments can be set up permanently. How to do this is explained in [17.5 Permanently Specify X_RUN Parameters](#).

Designed for use by application developers, X_START is a simple utility that prompts for X_RUN parameters. Refer to [17.1 Using X_START as a Front End to X_RUN](#) for details.

17.3 X_RUN Parameter Summary

These parameters have been listed in alphabetical order so that you can more easily find them.

If a parameter is specified more than once, then the last value processed is used.

If executing X_RUN on IBM i, only the PROC= parameter is required.

When used, the parameter must be followed by an = sign and the value, without any spaces.

Some of these parameters are very specialized and you will find details about them by following the link.

Param.	Meaning / Values	Req	Default Value
ASPW=	Application Server Password	No	Refer to 17.4.1 User ID a Password Default Values
ASUS=	Application Server User	No	Refer to 17.4.1 User ID a Password Default Values
BTN2=	Function key to simulate when mouse button 2 is double clicked. Not supported on Linux.	No	Prompt key
BTN3=	Function key to simulate when mouse button 3 is double clicked.	No	Prompt key
CDLL=	The name of the .DLL that should be used for communications to an attached server. Note: LCOMGR32.DLL must be used when connecting to a Visual LANSA Server.	No	The CMTH parameter is the relevant default value for both development and execution connections to servers) when the Visual LANSA system is installed.
CIPH=	The symmetric cipher to use when LANSA calls OpenSSL. This value is case sensitive. Possible values are listed in the OpenSSL documentation at www.openssl.org/docs . The value is one that is accepted by the		The default is aes-256-cb No other values are currently supported or tested. Use at your own risk.

EVP_get_cipherbyname API.

- CMTH= The communication method that should be used for conversations with any attached server. No The CMTH parameter is the relevant default value for both development and execution connections to servers) when the Visual LANSAsystem is installed.
- T = Native TCP/IP
- The C and T values are identical and can be used interchangeably. They are provided to allow you to more easily remember which value to use.
- Note: C or T must be specified when connecting to a Visual LANSAServer.
- DASO= Data Area Storage Option parameter. It may be set to 'D' or 'F'. No D is the default value.
- D indicates that data area value storage and locking should be emulated using table LX_DTA and the standard LOCK_OBJECT/UNLOCK_OBJECT logic.
- F indicates that data area value storage and locking should be emulated using flat operating system files stored in the same directory as any relative record number assignment files (refer to the RPTH= parameter). Locking and unlocking is achieved by using the appropriate operating system facilities for low level file access.
- Refer to the *LANSASApplication Design Guide for Guidelines, Rules and Limitations* that apply when using the DASO=F option.
- DATF= Date format to be used. Allowed values are DMY, MDY and YMD. If running in SuperServer mode to an IBM i server, this causes the job on the IBM i server to be run with this date format No X96SDF column from L2

which may differ from the IBM i system date format, QDATFMT.

DATS= Is used to specify from where to retrieve No S is the default value.
the date and date format.

S: Specifies that the date and date format are to be retrieved from the system values

J: Specifies that the date and date format are to be retrieved from the job attributes

Note:

IBM i only and is equivalent to the DATE_SRCE parameter of the LANSA command.

DBCL= Database Connection Level No 2

1. Handles database connections as in Visual LANSA V11.3 and earlier. Note that DBCF flags may still effect this connection, but they are not supported. Therefore, do not attempt to use DBCF flags when DBCL=1.

2. New database connection logic and support for DBCF flags.

DBCF= Database Connection Flags No CT_INTEGRATED_LOG

This option has been included for future flexibility, but currently is not supported. For further details refer to [17.4.2 DBCF Flags](#).

When this parameter is set on the X_RUN command line or equivalent, it only affects the main LANSA database, including SUBMITTED jobs and SuperServer jobs. It does not affect PC Other Files.

This parameter is unusual in that it can

be specified many times. Each setting takes the form: DBCF=<flag>:[Y/N], where the flag is one of the values specified in [17.4.2 DBCF Flags](#) and Y sets the flag on and N sets it off. For example,
 DBCF=CT_INTEGRATED_LOGON:Y

<p>DBCC= Cursor Concurrency ODBC default = 1 - SQL Server only Refer to LANSA and SQL Server - Configuration Options in the Tips and Techniques on the LANSAs web site.</p>	<p>Defaults to Read Only (1</p>
<p>DBCT= Cursor Type: ODBC default = 0 - SQL Server only. Refer to LANSA and SQL Server - Configuration Options in the Tips and Techniques on the LANSAs web site.</p>	<p>No Defaults to Static cursor (</p>
<p>DBHT= Details used, such as the Computer Name and Port, for debugging.</p>	<p>No The DBHT parameter is (from the LANSAs Setting Refer to Debug in the Vis LANSAs User Guide. You would normally not this parameter.</p>
<p>DBID= Database Id When executing X_RUN on IBM i, this parameter is not needed. Refer to 17.4.3 DBID, DBUT, DBII and DBIT Parameters for details</p>	<p>No The default is to *LOCA IBM i and LX_LANSAs on othe platforms.</p>
<p>DBII= Internal/Repository Database Identifier. Refer to 17.4.3 DBID, DBUT, DBII and DBIT Parameters for details</p>	<p>No Defaults to the same valu DBID=, so if they are the you do not need to specif DBIT</p>
<p>DBIT= The type of dictionary/repository database specified in the DBII=</p>	<p>No The default values are: MSSQLS for Windows</p>

parameter. If the DBII= parameter is not supplied, the DBUT database type will be used.

ODBCORACLE for Linux

Refer to [17.4.3 DBID, DBUT, DBII and DBIT Parameters](#) for details

- | | | | |
|-------|--|----|--|
| DBLK= | Database Lock Timeout in seconds. Setting this parameter overrides the LOCK_TIMEOUT setting in X_DBMENV.DAT for ALL databases. There are database-specific settings related to LOCK_TIMEOUT apart from the timeout itself, so it is imperative that you review 17.8 Lock Timeout . | No | The default is 0
A value of 0 indicates the no timeout. |
| DBMR= | Enable MARS - SQL Server only. Refer to LANSA and SQL Server - Configuration Options in the Tips and Techniques on the LANSA web site. | No | Defaults to No. |
| DBSA= | When using Adaptive Server Anywhere, after 240 minutes of no activity from a client, by default, it is disconnected. This can cause a problem if connections are idle for long periods. To ensure there is activity on all open connections, they are periodically activated. This argument specifies how often this happens. The value is specified in minutes. | No | 10 |
| DBSP= | Set savepoint - SQL Server only. Refer to LANSA and SQL Server - Configuration Options in the Tips and Techniques on the LANSA web site. | No | Defaults to No. |
| DBSS= | May be used to adjust the maximum number of reusable SQL statements that | No | The default is 50. |

are cached for reuse.
For details, refer to [17.4.4 DBSS
Parameter - Performance Tuning](#).

DBTB= Trim DBCS Blanks. No Y
'Y' = Yes or 'N' = No.

When DBTB=Y, when it receives focus, any input capable alphanumeric shift J field presented by a Function or Component will have DBCS blanks trimmed from it. There is no change to the behavior when such a field loses focus.

Examples of controls that would exhibit this behavior include input capable fields in browse lists and in RDMLX list-type controls such as grid and list view.

This does NOT include fields presented by Active-X controls (or any COM object), unless the Active-X Control is generated by LANSA.

DBTC Attempt Database Trusted Connection before userid/password connection. No No

When this is set to 'Y', then a trusted connection is attempted before using a User ID and password to establish the connection.

If DBTC is set to 'Y', DBCL is automatically set to 2.

If DBTC is set to N, a log in using the Usr ID and Password is attempted.

DEBUG= Turn on debugging No 'N'
'Y' (Yes) or 'N' (No)

DBUS= User name for the database login. When No USER= argument. Refer
executing X_RUN on IBM i, this [17.4.1 User ID and Passv](#)

parameter is ignored.

On other platforms, after logging on to the database, the value of DBUS is changed to reflect what was required to logon to the database. For example, if trusted connections are being used then SQL Server will return an empty value for user ID. This value is then assigned into DBUS. Therefore when GET_SESSION_VALUE is used it may be empty. It is also altered by the default behavior.

Default Values.

DBUT=	The type of user database specified in the DBID=parameter. Refer to 17.4.3 DBID, DBUT, DBII and DBIT Parameters for details	No	The default values are: MSSQLS for Windows ODBCORACLE for Linux
DELI=	Delete packages from the host monitor after installation. Allowable values for this parameter are Y (Yes) and N (No).	No	Y
DEVE=	User is a developer 'Y' (Yes) or 'N' (No)	No	'N'
DPTH=	Directory in which emulated IBM i data queues are to be created / accessed.	No	<sysdir\x_ppp (where pp is partition name).
DRIV= or ROOT=	The path containing Visual LANSA. This parameter is no longer required in Windows environments.	No	If Linux then '/lansa', else path from which X_RUN was executed, provided it contains 'X_LANSA'
EDLC=	The parameter only applies to DBCS. It controls whether any DBCS string being entered or assigned is checked for length compatibility with EBCDIC based DBCS systems that use the shift in and shift out characters.	No	Y

Allowable values for this parameter are Y (Yes) and N (No).

Use EXTREME CAUTION when turning this option off (value N). This will allow you to store DBCS strings that are fundamentally incompatible with EBCDIC shifted systems. A later design change to, say, a client/server application, that involves an EBCDIC server (such as an IBM i) may cause severe DBCS string storage/truncation problems.

EXCH=	Exchange file name (Designed for internal LANSA use only)	No	Null
EXPM=	Name of file to contain *LIMPORT export messages.	No	import.log in temporary directory.
EXPR=	Path that contains the file(s) for *LIMPORT to import. If the value of this parameter is the special value 'QDLS\<folder>', then the import will be attempted via a direct connection to the IBM i N.B. the parameters PSLU and PSPW must be specified (as a minimum).	No	The standard File Open dialog is presented.
EXPS=	Action to take when importing object security records. D = Delete: all existing security records for an object will be deleted before import. R = Replace: an existing user security record will be replaced with a matching incoming record. A = Append Only: all existing user security records will be retained, matching incoming records will	No	D

generate a duplicate warning.

- FATL= Y= forces the display of fatal errors in client-side applications when QUET=Y. Refer also to [17.4.13 QUET & FATL Parameters \(Quiet Mode of Operation & Fatal\)](#) for further information.
- FLDX= Interpret Numeric Keypad. No Y
Enter Key as a Field Exit key (i.e. The Tab Key).
This will occur on all controls except push buttons. That is, entry fields, check boxes, radio buttons and all lists.
'Y' (Yes) or 'N' (No)
- FORM= Form name No Null
- FUNC= Function name No Null
- FXQF= Force *.XQ* flat, read-only, repository files into a Visual LANSA environment using this parameter set to FXQF=*ALL. Refer to [17.4.5 FXQX Parameters](#) for information.
- FXQM= Use this parameter to control the maximum number of flat files that are kept concurrently open. No
Refer to [17.4.5 FXQX Parameters](#) for information.
- GUSR= Group User Name No *NONE
- HELP= Specifies the run-time help system to be used. No STD for Windows.
OLD indicates old-style two-window help. OLD (forced) for all other operating systems.
STD indicates the default single-window help with tab control for Contents and Index.
WIN is reserved for Windows Help.

This is no longer to be used for new applications.

HTM is reserved for HTML Help.

- HLPF= Specifies the help file to be used for HTML Help (HELP=HTM). No Depends on value of HEL
For WIN, the default is th
Visual LANSA Help File
For HTM it is the HTML
version of the Visual LANSA
Help File.
The default location for h
files is the
X_LANSA\EXECUTE\
<language> directory.
- Note:**
From Windows version Vista onwards, the viewer for HLP files will no longer be distributed by Microsoft.
For Windows Help, the filename is specified minus the language code and file-type extension. For example, HLPF=MYHELP might be expanded to MYHELPFRA.HLP.
For HTML Help, the filename is minus the file-type extension. For example, HLPF=MYHELP would be expanded to MYHELP.CHM.
For HELP=STD or HELP=OLD, the value of this parameter is ignored.
This means that when you use X_RUN HELP=WIN
HLPF=C:\TEMP\MYHELP, the actual file being looked for will be named C:\TEMP\MYHELPFRA.HLP (if LANS-FRA is used) or C:\TEMP\MYHELPENG.HLP (if LANG-ENG is used).
- HSKC= Enables *High Speed Key Checking*. This No
feature should only be used for
Windows or Linux platforms.
For information refer to [17.4.6 HSKC Parameter](#).
- ICWD= Change working directory on startup to No N
x_lansa. Only supported on IBM i.
Y = Yes N = No.

INIT=	Specifies a function to be automatically executed on application startup. Refer to 17.4.7 INIT and TERM Parameters for information.	No	
ITxx=	Trace Parameters. For information, go to 17.4.8 ITxx - Trace Parameters .	No	
JOB=	Job Name	No	Null
LANG=	Language. When executing X_RUN on IBM i, this parameter is not necessary.	Yes	The default is the partition default language from the LANSAs command.
LDAV=	LDA (local data area) file name	No	Null
LOCK=	Y indicates that Object Locks should be obtained when executing a LANSAs object. This is a read-only lock which blocks requests to obtain exclusive access to the object. For example, compiling an object requires exclusive access. Thus a Form cannot be compiled whilst it is being executed. LOCK=Y should only be used in a development environment. It has no purpose in a deployed environment.	No	
LOGO=	Indicates whether to show a logo indicating the version and date of Visual LANSAs system being used. Allowed parameter values are Y and N.	No	N
LPTH=	Fully qualified root directory for storage of BLOB and CLOB disk files. Requires an ending '\	No	Refer TPTH
MENU=	This parameter determines whether new icon and bitmap style menus and action bars should be used.	No	Y

MODE=	'T' (interactive) or 'B' (batch)	No	B for Linux otherwise 'I'.
ODBA=	Deprecated. Number of database connections now automatically determined by LANSA.		
ODBI=	Used to specify the transaction isolation level for all ODBC database connections. See ODBI Parameter for details.	No	Default is 0 (zero).
PARM=	Parameter file name (Designed for internal LANSA use only)	No	Null
PART=	Partition When executing X_RUN on IBM i, this parameter is not needed.	Yes	The default is to SYS fro LANSA command.
PBCM=	Specifies the color to be used for Column Headings. Values are G=Green, W=Black, R=Red, T=Turquoise, Y=Yellow, P=Pink, B=Blue. Invalid values are ignored.	No	B
PBFP=	Specifies the color to be used for Field Prompts (i.e. Labels and Descriptions). Values allowed are G=Green, W=Black, R=Red, T=Turquoise, Y=Yellow, P=Pink, B=Blue. Invalid values are ignored.	No	W
PPTH=	Fully qualified directory for report files if PRTR=*PATH. Requires an ending '\'.	No	x_lansa\x_ppp\ on the DRIV=path where ppp=partition.
PROC=	Process name	Yes	
PROG=	This parameter allows you to rename the X_RUN.exe file while allowing the SUBMIT command to work correctly.	No	

Refer to [17.4.11 PROG Parameter](#) for details.

PRTR=	Printer port name (e.g. LPT1, LPT2) or special value *PATH which indicates reports are to be output to a file rather than a printer. For Linux this should be the same dest name which would be used by the lp command. N.B. Use of PRTR with any value other than *PATH is deprecated for Windows and is no longer supported. Existing applications may continue to function correctly, but LANSAs does not warrant its use. Use the WPxx parameters instead.	No	Linux: *PATH Windows: LPT1
PSPW=	Primary Password for Server and Client	No	PSPW= argument. Refer to 17.4.1 User ID and Password Default Values details.
PSTC=	Specifies that Windows Authentication is used. Refer to 17.4.1 User ID and Password Default Values for details.	No	Default is N.
PSUS=	Primary Server User.	No	Refer to 17.4.1 User ID and Password Default Values details.
PSWD=	Password for the database login. When executing X_RUN on IBM i, this parameter is ignored.	No	PSPW= argument. Refer to 17.4.1 User ID and Password Default Values for details
PSxx=	Please note The PSxx parameters are primarily provided to aid developers in testing applications in SuperServer mode. Refer to 17.4.12 PSxx Server Parameters for the parameters available in the PSxx range.	No	

QCHK=	Indicates approximately how long an active monitor should wait before checking the job queue. Refer to Additional Job Queue Monitor Parameters for details.	No	Default is 10 seconds
QHLD=	Indicates how long a held job queue monitor should wait before rechecking the job queue directory for release instructions. Refer to Additional Job Queue Monitor Parameters for details.	No	Default is 30 seconds
QENC=	Specifies that the job details are to be encrypted before placing them on the queue. Refer to Encrypting the Job Queue Details for details.	Yes	Default is N
QUET=	Used to force a batch job into a quiet mode when normal error and status reporting activities are suppressed. Refer to 17.4.13 QUET & FATL Parameters (Quiet Mode of Operation & Fatal) for details.	Yes	QUET=N is the default.
RNDM=	Render Mode. H = Hardware, S=Software	No	Default value is H.
RNDR=	Render Style. W = Win32, M = Mixed, X = DirectX	No	Default value is W.
RPTH=	Fully qualified directory for RRNO (relative record number) files. Requires an ending '\'. Note that if Visual LANSA is installed on a server, then this path must be located on the server.	No	x_lansa\x_ppp\ on the DRIV=path where ppp=partition.
RRNA=	The number of RRN (Relative Record Number) assignments that are to be pre-allocated when inserting data into a table.	Yes	The default value is 1.

Refer to [17.4.14 RRNA and RRNB Parameters - Performance Tuning](#) for details.

- | | | |
|-------|---|--|
| RRNB= | Specifies whether Windows operating system file buffering is to be used when accessing the RRN or *AUTONUM data area assignment files. Specify this argument as Y or N.
Refer to 17.4.14 RRNA and RRNB Parameters - Performance Tuning for details. | The default is N. |
| TASK= | Task identifier to be used when executing applications that are performing development tasks (such as those using the specialized Built-In Functions). | No Null |
| TERM= | Specifies a function to be automatically executed on application shutdown.
Refer to 17.4.7 INIT and TERM Parameters for information. | |
| TPTH= | Fully qualified directory for temporary files. | No Refer to 17.4.15 TPTH Parameter for how TPTH derived. |
| UDEF= | User Defined Parameter.
256 byte alpha to be used to pass information into LANSAs on the command line.
Use GET_SESSION_VALUE and SET_SESSION_VALUE to get and set the value from RDML.
This parameter has no other use in LANSAs. It is provided purely for the use of RDML developers to more easily communicate between LANSAs jobs/processes. | No Empty/blank. |

UPCD=	MSI Upgrade Code. This is set automatically by the MSI Install. It is not recommended to alter it.	No	Null
USER=	User name for Server and Client. When executing X_RUN on IBM i, this parameter is ignored. USER is the actual IBM i user.	Yes	Refer to 17.4.1 User ID and Password Default Values defaults.
USEX=	Default User ID used when establishing connections to servers. When the user id is passed to X_RUN, it puts the exact case in the USEX parameter, the same as the SET_SESSION_VALUE now does.		USER is the basis of USI
WDTM=	Controls the Windows Desktop Heap in which the process is created. Refer to 17.4.16 WDTM Parameter (Windows Desktop Heap) for information.	Yes	WDTM=N is the default.
WPxx=	Refer to 17.4.17 WPxx - Windows Printing Extensions for these parameters.	No	
XAFP=	Indicates whether all fields defined in a function should be exchanged when prompting. Only use this option when position 499 of the system data area (DC@A01) of an associated LANSAs for the IBM i system is set to Y. Do NOT use this option in any other circumstances. Y = exchange all fields on prompt = do not exchange all fields.	No	N
XCMD=	Obscure certain details in the command line. Refer to 17.4.18 XCMD Parameter for details.	No	Default is N.

XENV= Sets a LANSAs environment variable. No
For details, refer to [Regional Settings](#)

17.4 X_RUN Parameter Details

For some commands you will need more information to use them. Where applicable, this information follows:

17.4.1 User ID and Password Default Values	17.4.6 HSKC Parameter	17.4.13 QUET & FATL Parameters (Quiet Mode of Operation & Fatal)
17.4.2 DBCF Flags	17.4.7 INIT and TERM Parameters	17.4.14 RRNA and RRNB Parameters - Performance Tuning
17.4.3 DBID, DBUT, DBII and DBIT Parameters	17.4.8 ITxx - Trace Parameters	17.4.15 TPTH Parameter
17.4.4 DBSS Parameter - Performance Tuning	17.4.9 ODBA Parameter	17.4.16 WDTM Parameter (Windows Desktop Heap)
17.4.5 FXQX Parameters	17.4.10 ODBI Parameter	17.4.17 WPxx - Windows Printing Extensions
	17.4.11 PROG Parameter	17.4.18 XCMD Parameter
	17.4.12 PSxx Server Parameters	

Also see

17.7 The .XQ* Files

17.4.1 User ID and Password Default Values

Notes:

The system variable *USER is derived from USER.

Database stamping attributes are derived from USER.

PSUS, ASUS, PSPW and ASPW will be automatically converted to uppercase when attempting to connect to an IBM i server system. For non-IBM i server systems their case is unchanged.

When PSTC=Y, that is you have directed LANSAs to use Windows Authentication (also called Trusted Connections), LANSAs will firstly attempt to obtain the value of USER by looking up the Windows authenticated user in file LX_FKU. If one is found then USER is assigned the value of the associated PC User ID. Any existing value of USER is overwritten. If one is not found, then the defaulting logic below is used. This lookup always occurs when PSTC=Y. That is, it's not just performed when USER is unassigned.

User Profiles

Order of Defaulting	X_RUN Param.	Description	Always Uppercase	Defaulting Order
1	USER	System User	Yes	1. DBUS 2. PSUS (upper case) 3. ASUS (upper case)
2	DBUS	Database User	Yes	1. USER 2. PSUS (upper case) 3. ASUS (upper case)
3	PSUS	Primary Server User	No	1. USER (exact case) 2. ASUS 3. DBUS (exact case)
4	ASUS	Application Server User	No	1. USER (exact case) 2. PSUS 3. DBUS (exact case)

Passwords

--	--	--	--	--

Order of Defaulting	X_RUN Param.	Description	Always Uppercase	Defaulting Order
1	PSPW	Database Password	Yes	1. PSPW (upper case) 2. ASPW (upper case)
2	PSPW	Primary Server Password	No	1. PSPW (exact case) 2. ASPW
3	ASPW	Application Server Password	No	1. PSPW (exact case) 2. PSPW

17.4.2 DBCF Flags

The following flags are mutually exclusive. They are tested in the following order. The connection will be established using the first flag found to be true.

1. **CT_OLD_STYLE_LOGON_SQLCONNECT:** Log on using User ID/Password. If that fails it's a fatal error.
2. **CT_DSN_ONLY_LOGON:** Only provide the DSN to ODBC presuming that the DSN completely defines all the connection parameters.
3. **CT_INTEGRATED_LOGON:** Perform an integrated Log on. Some databases require a special parameter to be passed to ODBC.
4. Use User ID and Password, if they have been passed in, if
 - the CT_DRIVER_PROMPT is specified and
 - the LANSA process is interactive and
 - it is running on Microsoft Windows.

Allow the ODBC Driver to prompt for further information if the information provided in the connection string and ODBC DSN are insufficient to establish a connection to the database.

In all other circumstances the CT_DRIVER_PROMPT is ignored. Further connection behavior is dictated by the connection algorithm being used.

The only purpose of using these flags is to speed up the connection by ensuring that the first connection succeeds. For example, if you know an integrated Logon will be the only connection you want to succeed, specify DBCF=CT_INTEGRATED_LOGON:Y and DBCF=CT_AUTHENTICATION_ERROR_IS_FATAL:Y. This will cause the first connection to be integrated and a fatal error will occur if that connection is unsuccessful.

Other DBCF Flags

CT_AUTHENTICATION_ERROR_IS_FATAL: An authentication error by default is not fatal. Setting this option on ensures that the first connection will either succeed or there will be a fatal error. This is similar behavior to LANSA Version 11.3.

CT_DISPLAY_AUTHENTICATION_PROMPT: When an authentication error occurs, display a message to the user. Typically used in conjunction with CT_DRIVER_PROMPT to cause ODBC to continually re-prompt the user for

connection information whilst there is an authentication error and cancel has not been clicked. These options were used with PC Other Files in Version 11.3.

CT_DRIVER_PROMPT: Ask the ODBC driver to prompt for more information if the connection information it has is insufficient to obtain a connection.

CT_ALL_FLAGS_ON: Sets all the flags on or off. Typically used to ensure that any preceding commands are neutralized by using

DBCFC=CT_ALL_FLAGS_ON:N. It does not make sense to use

DBCFC=CT_ALL_FLAGS_ON:Y, although all flags will be set on if it is used.

Also see

[Connection Algorithm](#)

Connection Algorithm

When DBCL = 1, the connection algorithm is to Log on to the main database using the supplied User ID and Password (Using ODBC API SQLConnect). If the Log on fails, LANSA exits with a fatal error. Files in Other databases are logged on with the supplied connection information and if that fails, the user is prompted for further database connection information (Using ODBC API SQLDriverConnect).

When DBCL = 2, the database Log on goes through a number of default attempts to log on using various different parameters. PC Other Files is essentially the same, except DBCF flags do not affect it and if none of the default connection methods work, it finally prompts for connection information from the user, as it does in LANSA V11.3.

Its important to note that specifying a DBCF flag doesn't alter the steps in the [SQL Anywhere](#) algorithm, but the behavior of each step may be changed if the DBCF flag has a higher precedence than the kind of connection being attempted in a particular step. For example, if CT_INTEGRATED_LOGON is set, then step 3 of the SQL Anywhere connection will behave the same as a trusted connection because trusted connection has a higher precedence than using User ID/Password (for precedences, refer to [17.4.2 DBCF Flags](#)).

The DBCL = 2 algorithm is described for:

- [SQL Anywhere](#)
- [Oracle](#)
- [SQL Server](#)
- [Other Databases](#)

SQL Anywhere

1. Attempt a connection using the parameters as passed in.
2. If DBTC = Y, attempt a trusted connection.
3. Attempt Logging on using the User ID and Password supplied, if any. This is for backward compatibility.
4. Connect just using the DSN. This presumes that the ODBC DSN contains all the connection information required.
5. Attempt to connect using DBA/SQL. This will be the most common connection made with new installations.
6. If it's a PC Other File, prompt the user repeatedly until a connection is successfully made or the dialog is cancelled. (This applies only to interactive LANSA processes running on Microsoft Windows.)
7. Fatal error.

Oracle

1. Attempt a connection using the parameters as passed in.
2. If DBTC = Y, attempt a trusted connection.
3. Attempt Logging on using the User ID and Password supplied, if any. This is for backward compatibility.
4. If it's a PC Other File, prompt the user repeatedly until a connection is successfully made or the dialog is cancelled. (This applies only to interactive LANSA processes running on Microsoft Windows.)
5. Fatal error.

SQL Server

Microsoft SQL Server has only ever fully supported trusted connections. Specifying a User ID and Password may work, but it has never been a supported feature. As such, there is no change to the database Log on to an SQL Server database, except that DBCF flags may affect the type of Log on performed.

1. Attempt a connection using the parameters as passed in.
2. If it's a PC Other File, prompt the user repeatedly until a connection is successfully made or the dialog is cancelled. (This applies only to interactive LANSA processes running on Microsoft Windows.)
3. Fatal error.

Other Databases

There is no change to the Log on to other databases, except that DBCF flags may affect the type of Log on performed. The default connection algorithm is as follows:

1. Attempt a connection using the parameters as passed in.
2. Prompt the user repeatedly until a connection is successfully made or the dialog is cancelled. (This applies only to interactive LANSA processes running on Microsoft Windows.)
3. Fatal error.

17.4.3 DBID, DBUT, DBII and DBIT Parameters

You use the DBID= (Database/Source Identifier) parameter with the X_RUN command to nominate the name of the database/source to which you wish to connect your application.

There are two database identifier parameters:

DBID User Database Identifier. This is the database where your application tables reside (e.g. CUSTMST).

DBII Internal/Repository Database Identifier. This is the database where the Visual LANSA dictionary/repository resides (e.g. LX_F03 - Field Definitions).

DBII= defaults to the same value as DBID=, so if they are the same you do not need to specify DBII.

Database Type

DBID= and DBII= have parameters DBUT and DBIT to indicate the type of database being used.

Note: If DBID= and DBII= are different, and if the special values *ANY, *NONE, *AS400 or *OTHER are not being used, then the DBID and DBII must have identical Visual LANSA repository and object definitions. For example: DBII=LX_LANSA, DBID=LX_USER, DBIT=MSSQLS (if provided, DBUT must be the same as DBIT), both LX_LANSA and LX_USER contain the same LANSA object definitions. Therefore, whenever an object is added or modified, it needs to be exported to other database.

The parameters DBIT and DBUT **must** specify the same type of database. For example DBIT=MSSQLS and DBUT=MSSQLS. It would NOT be valid to use this: DBIT=MSSQLS and DBUT=SQLANYWHERE.

DBUT The type of user database specified in the DBID=parameter. The default values are:

MSSQLS for Windows

ODBCORACLE for Linux.

DBIT The type of dictionary/repository database specified in the DBII= parameter. If the DBII= parameter is not supplied, the DBUT database type will be used.

The default values are:

MSSQLS for Windows
ODBCORACLE for Linux.

The DBUT= and DBIT= values are important because they link the database/source named in the DBID= and DBII= to the database characteristics.

The database characteristics are defined in the "x_dbmenv.dat" (Database Environment Definition File) which is described in [The X_DBMENV.DAT File](#).

The ability to link the database/source that you specify (in, say, DBID=) to a database type (specified in DBUT=) is vital.

For example, if you specify DBUT=MSSQLS then the entire set of error messages/return codes issued by the DBMS is different to when you specify DBUT=SQLANYWHERE. These DBMS specific variations are defined in the "x_dbmenv.dat" file.

Special Values DBID=*NONE and DBII=*NONE

Visual LANSA supports the use of special values *NONE in the DBID= and DBII= parameters.

The special value *NONE indicates that no local (or connected server) database is available to the Visual LANSA application(s).

By using this option, Visual LANSA applications can be made to execute without the need to have any form of SQL/ODBC style database installed/available locally.

When using DBID=*NONE all process, function and file level security checking is disabled.

If you use *NONE with DBID or DBII, you will also use the XQ files which are described in [17.7 The .XQ* Files](#).

Normally you would only use this option on end user PCs or when testing an application.

To use any form of Visual LANSA or Visual LANSA development facility you must have an accessible repository database.

Special Value DBID=*ANY, DBID=*AS400 or DBID=*OTHER

Visual LANSA supports the use of special value *ANY, *AS400 or *OTHER in the DBID= parameter.

These special values indicate that no local (or connected server) database is available to the Visual LANSA application(s) and that the first function or component invoked should automatically connect to a specified server.

When using DBID=*ANY, *AS400 or *OTHER you will use the .xq* files

described in [17.7 The .XQ* Files](#) as well as a series of PSxx= parameter values that define the characteristics of the PS (Primary Server) that you wish to automatically connect to. Refer to [17.4.12 PSxx Server Parameters](#) for the PSXX parameter requirements.

Using DBID=*ANY, AS400 or *OTHER is exactly the same as using DBID=*NONE, except that you will get an automatic connection to a nominated server system.

By using this option, Visual LANSA applications can be made to execute without the need to have any form of SQL/ODBC style database installed/available locally.

Normally you would only use this option on end user PCs or when testing an application. When using Visual LANSA for development, you must have an accessible repository database.

Valid DBID= and DBII= Parameter Settings and Recommendations

These are the valid and recommended DBID= and DBII= settings for various client (Windows)/server environments:

Environment	Comments	DBID=	DBII=
Full Client/Server to a Server:	All applications work directly to the Server DBMS and no local DBMS is required/available.	*ANY *AS400 *OTHER	*NONE
Mixed Client/Server:	Most applications work directly to the Server's DBMS and limited local DBMS access is required.	<name>	*NONE
Heavily Client side:	Most applications work directly to the local DBMS and limited or no Server DBMS access is required.	<name>	*NONE

17.4.4 DBSS Parameter - Performance Tuning

This X_RUN parameter may be used to adjust the maximum number of reusable SQL statements that are cached for reuse.

DBSS=0 will disable caching, which is not recommended, as it will generally result in performance degradation unless no SQL statements are reused.

DBSS=50 is the default.

Also see

[17.4.14 RRNA and RRNB Parameters - Performance Tuning](#)

17.4.5 FXQX Parameters

FXQF Parameter

When you have installed a set of *.XQ* flat read-only repository files into a Visual LANSA environment you can force them to be used by specifying the parameter FXQF=*ALL. Normally use of the read-only tables only occurs when the DBII= (Database Internal Identifier) parameter is set to *NONE. In some situations you may need to have DBII=LX_LANSA (for example) so that access to internal tables such as LX_FOL (Object Locks), LX_DTA (Data Area Emulation) and LX_F80 (Saved Lists) is still possible. You can do this by using the combination of parameters DBII=LX_LANSA and FXQF=*ALL thus gaining access to internal non-static repository tables and also forcing the high performance *.XQ* read only repository tables to be used.

FXQM Parameter

When flat, read-only repository files are being used because parameter DBII=*NONE or because parameter FXQF=*ALL is being used, then the logic to read information from the flat tables is optimized to keep as many flat files open as possible. In some situations the operating system may limit how many flat files can be concurrently open. Use this parameter to control the maximum number of flat files that are kept concurrently open. If you do not specify this parameter it defaults to 60. Otherwise set it to a value in the range 4 to 256. If you receive an error message indicating that the FXQM parameter needs to be reduced try setting it to FXQM=59, then FXQM=58, etc., until the message is no longer issued.

Also see

[17.7 The .XQ* Files](#)

17.4.6 HSKC Parameter

The HSKC=Y parameter may be specified (or defaulted) on the X_RUN command

When this value is used, "High Speed Key Checking" is enabled. This feature should only be used for Windows or Linux platforms.

When this parameter is enabled, all eligible FILECHECK RDML commands and OAM based "File Lookup Checks" will automatically track key values that have previously been found to exist in the associated SQL table(s). These tracked key values are held in memory and are much faster to (re)access second and subsequent times.

By using the HSKC= parameter selectively on relatively static decode and validation tables you can significantly improve the performance of an application.

For example, there may exist an SQL table called COMPANY that contains the identifiers of all valid companies within your application. Many other SQL tables such as CUSTOMER, ORDER, PRODUCT, etc. may have referential integrity checks against the COMPANY table that are invoked whenever information is inserted, updated or deleted.

When performed under SQL, these lookup checks into COMPANY are relatively expensive, but with the HSKC=Y parameter, all (re)checks of the same key value (i.e. the company identifier) against the COMPANY table are much cheaper and faster to execute.

Typically, eligible files include relatively static tables such as "Company Names", "Zip Codes", "Currency Codes", etc. that are used extensively for validation.

Before using this facility you should be aware of the following:

- You must nominate the names of all Physical Files/SQL tables that are eligible for this type of processing in a simple text file named X_HSKC.DAT in the main x_lansa directory of your Visual LANSA system. This file is used at execution time, so it must be present on development and execution systems.
- File X_HSKC.DAT is a simple text file that can be created and edited with most standard source file editors. Specify the name of only one file/table per line. Upper or lower case characters may be used in the names. Only the physical file/SQL table names need be specified. All logical views of the table

are implicitly included when the physical file is named. The name of the file must be the first thing on the line. Leading blanks are significant and are not ignored.

- No checking of any type is done on the names specified in X_HSKC.DAT. If you enter an invalid or incorrect file name it will be accepted, and the correct file will not be subject to the HSKC=Y effects.
- If you use the HSKC=Y parameter and attempt to do a file lookup check for file X_HSKC.DAT, if it does not exist in the x_lansa directory, a fatal error will result, terminating the application.
- Updates or deletes made to a table from within the current X_RUN process/session cause all tracked details for the table to be dropped (thus restarting the tracking process again).

However, you must note that only updates and deletes issued from within the current X_RUN process/session cause this to happen. Updates and deletes issued by other network users, or to an IBM i server (via LANSASuperServer), or via other X_RUN processes, do not cause the tracked key data to be dropped from the current X_RUN session.

- The amount of key data that can be tracked is limited to the memory size of your computer. Tracked key data uses the aggregate byte length of the key involved + 2 bytes for each key value that is being tracked.
- High Speed Key Checking is not implemented for Char, String, Nchar or Nvarchar keys.

High Speed Key Checking for OAM based 'File Lookup Checks' is relevant for Windows and Linux platforms only. That is, if the server is a Windows or Linux server then this parameter is relevant. It is not relevant for a Visual LANSAS to IBM i set up. However, for FILECHECK RDML commands, the Visual LANSAS IBM i set up is fine. If you want to speed up the static files on the IBM i, then you should use the 'High Speed table' option on these files. This should yield a significant improvement in performance.

17.4.7 INIT and TERM Parameters

The purpose of the INIT= and TERM= parameters is to allow you to initialize or clean up your application automatically.

These two parameters are entirely independent of each other: you may specify one, both or neither.

INIT= (Startup Function) specifies a function to be automatically executed on application startup.

TERM= (Shutdown Function) specifies a function to be automatically executed on application shutdown.

Once either parameter has been specified, it is automatically passed to batch jobs so that they too will automatically run the same function on startup or shutdown.

For example, you might create a startup function CONNECT for a SuperServer application. This function could execute the BIFs DEFINE_ANY_SERVER (or DEFINE_OS_400_SERVER or DEFINE_OTHER_SERVER), CONNECT_SERVER and CONNECT_FILE with a standard set of arguments (such as blocking factor set to 1 on CONNECT_FILE). Not only would your main application connect to the Server automatically, so would any batch jobs.

Note: You should avoid the use of DISPLAY/REQUEST/POP_UP commands in INIT= and TERM= functions. Under some circumstances, these RDML commands (which require user interaction) may be inappropriate or may cause application failure.

17.4.8 ITxx - Trace Parameters

ITRO=

This parameter specifies whether the application is to produce a trace file. Specify Y to produce a trace file or N to not produce a trace file. Trace files are named x_tracennn.txt. The highest nnn suffix indicates the newest trace file. The production of trace files severely impacts application performance.

ITRL=

This parameter specifies the level of trace. Valid values are 0 to 9, where 0 provides the lowest level of detail and 9 the highest level of detail. This should not be changed unless requested by your Product Vendor.

ITRM=

This parameter specifies the maximum number of lines in the trace file. The maximum number you can enter is 999,999,999.

ITRC=

This parameter specifies the trace categories. It allows you to restrict the areas of LANSa that will generate trace messages. This should not be changed unless requested by your Product Vendor. Use of this value is described in [17.9 User Instructions for Microsoft Exception or Dr Watson](#). Multiple values can be specified at a time as one string, e.g. DBMUIM.

ALL	All categories
DBM	Database only
UIM	User Interface only
FUN	Standard Function only
PIM	Printer functions only
COM	Communications only
PDF	Platform Dependent Functions only
BIF	Built-In Functions only
PRO	Reserved
RDM	RDML only

RDX	RDMLX only
HEP	Heap Validation only

ITHP=

This parameter specifies the heap validation level. This should not be changed unless requested by your Product Vendor. Use of this value is described in [17.9 User Instructions for Microsoft Exception or Dr Watson](#). Note that it does not require ITRO=Y in order to validate the heap. Setting ITRO=Y will just add trace messages to the heap validation. They are often used in conjunction in order to provide detailed diagnostic information for use by your Product Vendor.

X	Use default as set in code (N for GA versions, G for internal debug versions)
N	No heap validation
G	Guard bytes and validate pointer
P	Validate pointer only
H	P + validate whole of heap that the pointer is in
A	H + validate all heaps
T	H + trace validations
Z	A + trace validations

17.4.9 ODBA Parameter

The ODBA= parameter has been deprecated. The number of database connections required is automatically determined by LANSAs.

17.4.10 ODBI Parameter

The ODBI parameter is used to specify the transaction isolation level for all ODBC database connections.

The default is 0.

Valid values are as per the table following. Any other values will be ignored and the default transaction isolation level will be used. Note that some ODBC drivers may not support all of the transaction isolation levels and may return an error when attempting to set the transaction isolation level.

Note: Careful consideration must be given when specifying this value as it affects all ODBC database connections for the executing application. This is an issue between you, the designer of the application, and your chosen database management system.

Value	Transaction Isolation Level	Meaning
0	SQL_TXN_READ_COMMITTED	Refer to Value 2 unless the default for the database is higher.
1	SQL_TXN_READ_UNCOMMITTED	Dirty reads, non-repeatable reads and phantoms are possible. This is the default for SQL Anywhere.
2	SQL_TXN_READ_COMMITTED	Dirty reads are not possible. Non-repeatable reads and phantoms are possible. This is the default for Oracle and SQL Server.
3	SQL_TXN_REPEATABLE_READ	Dirty reads and non-repeatable reads are not possible. Phantoms are possible.
4	SQL_TXN_SERIALIZABLE	Transactions are serializable. Dirty reads, non-repeatable reads and phantoms are not possible.
6	SQL Server only: SQL_TXN_SS_SNAPSHOT	Refer to LANSA and SQL Server - Configuration Options in the Tips and Techniques on the LANSAs web site.

Platform Specific Notes

IBM i:

- For files that are under [Commitment Control](#), a commitment definition must exist before any updates are made to these files. Refer to [Commitment Control](#) in the *LANSA for i User Guide* for more information.
- The ODBI parameter is ignored. If a commitment definition exists, LANSA sets the transaction isolation level appropriately for the lock level used when commitment control was started. Otherwise, the transaction isolation level is set to **Read Uncommitted**. Refer to *Isolation Level* in the *DB2 for i SQL Reference* for more information.

Also see

[Commitment Control](#) in the *LANSA Application Design Guide*.

17.4.11 PROG Parameter

This parameter is provided so that the X_RUN.exe file can be renamed and still allow the SUBMIT command to work correctly. There is no other support for renaming X_RUN.exe, for example in the deployment tool. You will need to do this yourself using batch files and the like to rename files.

Providing a value for PROG only changes the behavior of the SUBMIT command when using the Process/Function variant. Instead of starting up X_RUN.exe, the value provided for PROG will be used instead. A full path, up to 256 characters, should be provided to ensure the correct executable is used.

PROG supports quoted values for providing paths that contain spaces and is passed on to submitted jobs.

Client applications will automatically use the name of the executable used to start them when submitting jobs. In effect, the PROG parameter is automatically initialized to the name of the current executable. It is only server applications that need the PROG parameter specified.

17.4.12 PSxx Server Parameters

Please note these parameters are primarily provided to aid developers in testing applications in SuperServer mode. It is very strongly recommended that you use your own entry point function in production applications. This function should establish any required SuperServer user profile and connections details and then connect via the Built-In Functions described below.

An additional specialized set of parameters may be used on the X_RUN command to establish an automatic connection to a single server (called the "PS" or "Primary Server").

These parameters are specifically designed to be used in conjunction with the DBID=*ANY/*AS400/*OTHER parameter described earlier.

These parameters will be ignored unless one of the special values DBID=*ANY, DBID=*AS400 or DBID=*OTHER is used to trigger their use..

These specialized parameters directly equate to, and default like arguments to server Built-In Functions as described in the *LANSAs Technical Reference Guide*.

When connecting to a Server with DBID=*ANY, the following PSxx arguments can be used:

Param	Directly Equates to this Built-In Function:	Argument Number
PSLU	DEFINE_ANY_SERVER	2 (LU partner name)
PSCC	DEFINE_ANY_SERVER	3 (Commitment control)
PSEA	DEFINE_ANY_SERVER See PSEA Notes (Primary Server Exceptional Arguments)	4 (X_RUN exceptional arguments)
PSDL	DEFINE_ANY_SERVER	5 (Divert locks)
PSWM	DEFINE_ANY_SERVER	6 (Show please wait message)
PSEP	DEFINE_ANY_SERVER	7 (Server execution priority)

PSCT	DEFINE_ANY_SERVER	8 (Client to Server table)
PSST	DEFINE_ANY_SERVER	9 (Server to Client table)
PSPW	CONNECT_SERVER	2 (Password)
PSTC	CONNECT_SERVER. See PSTC Notes (Primary Server Trusted Connection)	3 (Use Kerberos Authentication)
PSUS	See PSUS Notes (Primary Server User)	
PSTY	See PSTY Notes (Primary Server Type) .	
PSRA	See PSRA Notes (Primary Server Route Authority)	
PSRR	See PSRR Notes (Primary Server Route Repository)	

The SSN (Symbolic Server Name) used when automatically connecting to a server with DBID=*ANY is always DTASERVER. This value cannot be changed.

When connecting to an IBM i with DBID=*AS400, the following PSxx arguments can be used:

Param	Directly Equates to this Built-In Function:	Argument Number
PSLU	DEFINE_OS_400_SERVER	2 (LU partner name)
PSCC	DEFINE_OS_400_SERVER	3 (Commitment control)
PSDB	DEFINE_OS_400_SERVER	4 (DBCS capable)
PSDL	DEFINE_OS_400_SERVER	5 (Divert locks)
PSWM	DEFINE_OS_400_SERVER	6 (Show please wait message)
PSEP	DEFINE_OS_400_SERVER	7 (Server execution priority)

PSCT	DEFINE_OS_400_SERVER	8 (Client to Server table)
PSST	DEFINE_OS_400_SERVER	9 (Server to Client table)
PSPW	CONNECT_SERVER	2 (Password)
PSUS	See PSUS Notes (Primary Server User)	
PSTC	CONNECT_SERVER. See PSTC Notes (Primary Server Trusted Connection)	3 (Use Kerberos Authentication)
PSTY	See PSTY Notes (Primary Server Type) .	
PSRA	See PSRA Notes (Primary Server Route Authority)	
PSRR	See PSRR Notes (Primary Server Route Repository)	

The SSN (Symbolic Server Name) used when automatically connecting to an IBM i server with DBID=*AS400 is always AS400. This value cannot be changed.

When connecting to a Server with DBID=*OTHER, the following PSxx arguments can be used:

Param	Directly Equates to this Built-In Function:	Argument Number
PSLU	DEFINE_OTHER_SERVER	2 (Server network name)
PSDL	DEFINE_OTHER_SERVER	3 (Divert locks)
PSWM	DEFINE_OTHER_SERVER	4 (Show please wait message)
PSEA	DEFINE_OTHER_SERVER See PSEA Notes (Primary Server Exceptional Arguments)	5 (X_RUN exceptional arguments)
PSTC	CONNECT_SERVER. See PSTC Notes (Primary	3 (Use Kerberos

	Server Trusted Connection)	Authentication)
PSPW	CONNECT_SERVER	2 (Password)
PSUS	See PSUS Notes (Primary Server User)	
PSTY	See PSTY Notes (Primary Server Type) .	
PSRA	See PSRA Notes (Primary Server Route Authority)	
PSRR	See PSRR Notes (Primary Server Route Repository)	

The SSN (Symbolic Server Name) used when automatically connecting to a non-IBM i server with DBID=*OTHER is always DTASERVER. This value cannot be changed.

When these parameters are used in conjunction with the DBID=*ANY/*AS400/*OTHER parameter, the first function or component to be invoked will act as if it used these Built-In Functions during its start up logic:

- DEFINE_ANY_SERVER, DEFINE_OS_400_SERVER or DEFINE_OTHER_SERVER to define the server as per the PSxx= parameter
- CONNECT_SERVER to establish the connection
- CONNECT_FILE to connect all files (file name "*" is used) to the server.

Note: The connection uses the default blocking factor on all files. This means that SELECT / UPDATE / ENDSELECT and SELECT / DELETE / ENDSELECT loops that are updating the last record read (i.e. the DELETE or UPDATE has no WITH_KEY or WITH_RRN parameter) may not process as expected. Refer to the CONNECT_FILE Built-In Function for more details of this problem and how to correct it. Another method of correcting this problem is described in [17.4.7 INIT and TERM Parameters](#).

If any of these actions fail then the function will fail during start up. This cannot be trapped. To use trapping, avoid using the PSxx= parameters and initially invoke your own start up function instead.

PSUS Notes (Primary Server User)

If this parameter is not specified, the user used to connect to the server is the same as that specified in the USER= parameter.

PSEA Notes (Primary Server Exceptional Arguments)

This argument must be enclosed in double quotes; for example:

PSEA="DBIT=*SERVER DBUT=*SERVER"

Refer to the definition of the [DEFINE_OTHER_SERVER BIF](#) in the *LANSATechnical Reference Guide* for further information on setting up X_RUN exceptional arguments.

PSRA Notes (Primary Server Route Authority)

Setting PSRA=Y indicates that authority checks should be routed to the server. The following notes apply to using the PSRA (or an equivalent) option:

It is recommended that you choose one of these methods to use this option:

- Put PSRA=Y into a profile file or,
- In your connection routine, put a USE SET_SESSION_VALUE (PSRA Y) command to set the value on the fly. This should be done before you define and make the connection or,
- In your DEFINE_XXXXXXXX_SERVER command, set the lock objects value to Z (which means route lock requests and route authority requests).

Things you should know about using PSRA=Y (or an equivalent option) include:

- If you are using authority checks to limit or restrict access to an initial process menu or action bar item and are creating a SuperServer connection by using the PSxx command line parameters, then remember that the SuperServer connection is not created until the first function is executed. Any initial process menu or action bar would be presented before the SuperServer connection is created. To resolve this issue create an entry point function that itself calls the initial process menu and then invoke it (instead of the initial process menu) from the command line.
- The use of PSRA=Y (or equivalent) has a performance impact because more trips to the server are required.
- Authority checks are optimized by being kept in memory, so if you change authority settings any active jobs/processes may not see the changes immediately (LANSA for i works the same way).
- If the user profile or the group profile you are using is QSECOFR, or the LANSA partition security officer, or the LANSA system owner then you always get access. No authority checks are actually made.
- Authority checks check authority. They do not check for existence and cannot be viably used to do this.
- Process, Function and File Security can be enabled in one of two ways. The recommended way is through System Maintenance. The alternative method is to modify [Regional Settings](#) in the X_DEFppp.H Definition Header File.
- If file security is disabled (which is the shipped default) then you will always

get access to files.

- If process security is disabled (which is the shipped default) then you will always get access to processes and functions.
- If function security is disabled (which is the shipped default) then you will always get access to a function.
- If you are running in SuperServer mode and have no accessible local database and are using PSRA=N (the default) then you will always get access. There is effectively no security when working in this mode.
- If a security check for a function is being made, and no security information at all exists for the function, then the authority of the owning process is "adopted". This logic (which has always existed) is designed to accommodate sites that suddenly turn on function level security (and thus have no security information available for their existing functions).

PSRR Notes (Primary Server Route Repository)

Setting PSRR=Y (the default) indicates that if repository data cannot be retrieved locally, a request should be sent to the server to retrieve the data.

Setting PSRR=N turns off this feature.

To turn off this option it is recommended that you:

- Put PSRR=N into a profile file

or

- In your connection routine, put a USE SET_SESSION_VALUE (PSRR N) command to set the value on the fly. This should be done before you define and make the connection.

If you use the DEFINE_XXXXXXXX_SERVER BIF to connect to the server, the lock objects value can be set to R (which means route lock requests, route authority requests, and repository requests). To turn off, routing repository requests, set the lock objects value to something other than R **AND** do one of the above recommendations to set PSRR to N.

Some examples of the information that can be automatically retrieved from the server repository are messages, partition definition, help text and object descriptions for the help index.

Things you should know about using PSRR=Y (or an equivalent option) include:

- The use of PSRR=Y (or equivalent) has a performance impact because more trips to the server are required. For best performance, repository information needs to be available locally, in the *.xq* files or in a local database.
- Automatic help generation data is **NOT** retrieved from the Server if it is not found locally.
- System definitions and partition language definitions are **NOT** retrieved from the Server if they are not found locally. However, this information will be set to defaults and processing will continue.
- Only **missing** information is retrieved from the Server during application execution, as the server request is only generated if the data is not found locally. Therefore, **changed** information will **not** be retrieved. If definitions have changed, they need to be redeployed to the client.

For further information on this feature, please refer to [17.7 The .XQ* Files](#).

PSTC Notes (Primary Server Trusted Connection)

When this parameter is specified, it is used to set the default value of *Use Kerberos Authentication* when calling the CONNECT_SERVER Built-In Function. For example, if PSTC=Y, then the CONNECT_SERVER Built-In Function's default will be to use Kerberos authentication.

PSTY Notes (Primary Server Type)

This parameter is only used for PROC=*LIMPORT/PLUGIN/REFRESH processing. It is used to set the server type as in the [Special Value DBID=*ANY, DBID=*AS400 or DBID=*OTHER](#), while allowing a local database to be specified with DBII=LX_LANSA. Note that PSTY=*OTHER and PSTY=*ANY are not currently supported for PROC=*LIMPORT/PLUGIN/REFRESH.

17.4.13 QUET & FATL Parameters (Quiet Mode of Operation & Fatal)

The QUET= parameter can be used to force a batch job into a quiet mode of operation so that the normal error and status reporting activities are suppressed. Using QUET=Y in an interactive process has no effect and the setting is ignored.

- Using QUET=Y in a batch job that fails suppresses the display of fatal error information unless you also set FATL=Y (see description following).
- If QUET=Y and FATL= N is set in a batch job that fails, details are logged to X_ERR.LOG in the usual way, but they are never actively displayed.
- Using QUET=Y in a batch job suppresses event log displays of status messages. Any process created will use the LANSAP Desktop Heap.
- You would normally set QUET=Y in a profile file or an environment variable rather than as a direct X_RUN command parameter.

QUET=N is the default.

FATL=

- If FATL=Y, the display of fatal errors (only) is forced in client-side applications when QUET=Y. This is its only effect.
- It does not affect Server or web jobs (it is only in x_uimms*.dll not x_usv.dll or x_u4w.dll).

Also see

[17.4.16 WDTM Parameter \(Windows Desktop Heap\)](#)

17.4.14 RRNA and RRNB Parameters - Performance Tuning

RRNA=

The number of RRN (Relative Record Number) assignments that are to be pre-allocated when inserting data into a table. The default value is 1, but any value in the range 1 to 5000 is valid.

Using a value such as 100 can substantially improve the performance of an application doing a large numbers of insert operations because 100 RRN numbers are (pre)allocated by a single access to the RRN assignment file, thus minimizing the number of times that the RRN assignment file needs to be accessed.

Using a value other than 1 may of course waste RRN values that are never assigned to an insert operation, thus leading to a faster rate of RRN value consumption than normal.

RRNB=

Specifies whether Windows operating system file buffering is to be used when accessing the RRN or *AUTONUM data area assignment files. Specify this argument as Y or N. The default is N.

Using Y will increase application performance but it increases the risk that the RRN or *AUTONUM data area assignment file contents may not be flushed to disk in the advent of a catastrophic system failure (eg: power failure) meaning that the contents may need to be correctly reset.

Also see

[17.4.4 DBSS Parameter - Performance Tuning](#)

17.4.15 TPTH Parameter

An example of what goes in the temporary path are trace files.

The temporary path evaluation has Linux and IBM i differences. First TPTH, ROOT, PROC, FORM, MODE and all the trace x_run parameters (e.g. ITRO) are evaluated.

This is how the temporary path directory is resolved:

1. TPTH x_run parameter, in any of the accepted locations for an x_run parameter.
2. TEMP environment variable.
3. TMP environment variable.
4. <sysdir> directory
 - On Linux: /tmp
 - On other platforms: <sysdir>
5. It should never get here, but if it does, a Fatal error occurs. As the temporary directory is required to write trace and error files, interactive jobs will show a message box and server jobs will output a message to STDOUT. On Windows the Listener can be run as a process rather than as a service to see these messages (lcolist -sstop; lcolist -c -d -x)

In the above list <sysdir> is the path of the x_lansa directory. For example, on Windows it could be c:\program files\lansa\x_win95\x_lansa. On Linux and IBM i it could be /home/LANSA_devpgmlib/x_lansa.

Enclosing double quotes and all trailing path separators and blanks are stripped before validating the directory.

If the directory does not exist then an attempt will be made to create it. If it fails then the next step is used.

The typical value for a temporary directory on Windows will be %TEMP% (typing that into Windows Explorer will take you to the directory). Note that server jobs typically use the user Local Server. Its %TEMP% value is not the logged on user's! For example, it may be c:\windows\temp. Process Monitor can be used on lcolist.exe to see the value of its TEMP environment variable.

The typical value for temporary directory on Linux and IBM i will be:
/lansa_devpgmlib/x_lansa/tmp

When the temporary directory is resolved, it is output to STDOUT.

The log directory contains x_err.log and export.log.

On Linux and IBM i, the log directory follows the same path as the temporary directory on Windows. That is:

1. LOGDIR environment variable
2. <sysdir>/log directory, if it exists or if it can be created
3. <sysdir>

17.4.16 WDTM Parameter (Windows Desktop Heap)

This X_RUN parameter controls the Windows Desktop Heap in which child processes are created.

WDTM is inherited by child processes.

WDTM=N is the default.

- WDTM=Y - the child process is created in the LANSAs Desktop Heap no matter where it is started from (i.e. Interactive Desktop Heap (IDH) or Service Desktop Heap (SDH)).
- WDTM=N - the child process is created in the Desktop Heap from which it started. Note, that if the current process was created in the LANSAs Desktop Heap then the child process is also created in the LANSAs Desktop Heap, but it is not controlled by LANSAs and therefore too many processes may be created in a LANSAs Desktop Heap. So, only set WDTM=N if the current process is the top most process. When a process is started by the operating system it will use either the IDH or SDH. In this case, setting WDTM=N is consistent with where a child process will be created - not in the LANSAs Desktop Heap.
- WDTM=I - the behavior is the same as for WDTM=N

For information about Desktop Heaps, refer to the the *Web Administrator's Windows Desktop Heap Management* in [Load Management](#) in the *Web Administration Guide*.

17.4.17 WPxx - Windows Printing Extensions

Visual LANSA supports extended printing options in Windows 32 bit environments.

Extended options are enabled and controlled by the WPxx= parameters on the X_RUN command. The WP component of the parameter name, identifies the parameter as being related to Windows Printing.

WPxx= parameters, like most X_RUN parameters, can be permanently set by using a profile file or an environment variable. Refer to [17.5 Permanently Specify X_RUN Parameters](#) for more details of how parameters can be permanently set.

The parameters used to enable and control Windows Printing are:

WPEN (Windows Printing Enabled)	WPDF (Windows Printing Font Dialog)
WPPN (Windows Printing Printer Name)	WPDF (Windows Printing Default Font)
WPPS (Windows Printing Setup File)	WPDS (Windows Printing Default Font Size)
WPPD (Windows Printing Print Dialog)	WPFO (Windows Printing Fixed Pitch Only)
	WPAS (Windows Printing Automatic Stretching)

Should you have problems with printing, try the [Questions and Answers](#) to see if the solution has been provided.

WPEN (Windows Printing Enabled)

WPEN=Y or WPEN=y enables Windows extended printing.

Using any other value disables windows extended printing.

This parameter is valid, but ignored, in all non-Windows 32 bit environments.

The default value is WPEN=N.

WPPN (Windows Printing Printer Name)

The Windows Printing Printer Name specifies the full name of the printer. If specifying a network printer, the domain name must also be included. That is, WPPN=\\domain\printer name. For example, WPPN=\\ourdomain\Epson Stylus COLOR 900.

The LIST_PRINTERS BIF returns the full printer name as required by this parameter.

This parameter is not passed to (inherited by) a server system. To print from a server in a client/server environment, retrieve the list of printers defined on the server by calling a function on the server that uses the LIST_PRINTERS BIF. The selected printer's name can then be exchanged back to the server when the print function is called. For example:

```
*****
```

```
** CLIENT SIDE FORM
```

```
*****
```

```
FUNCTION OPTIONS(*DIRECT)
```

```
BEGIN_COM ROLE(*EXTENDS #PRIM_FORM) CLIENTHEIGHT(240) C
```

```
* DEFINE_COM commands appear here to define controls on dialog.
```

```
DEF_LIST NAME(#PRNLIST) FIELDS(#PRN_NAME #PRN_LOC) TYPE(
```

```
EVTRoutine handling(#com_owner.Initialize)
```

```
SET #com_owner caption(*component_desc)
```

```
USE BUILTIN(SET_SESSION_VALUE) WITH_ARGS(USER MyUserid)
```

```
USE BUILTIN(DEFINE_ANY_SERVER) WITH_ARGS(MYSERVER Serv
```

```
USE BUILTIN(CONNECT_SERVER) WITH_ARGS(MYSERVER 'MyPas
```

```
USE BUILTIN(CALL_SERVER_FUNCTION) WITH_ARGS(MYSERVER
```

```
SELECTLIST NAMED(#PRNLIST)
```

```
ADD_ENTRY #LTVW_1
```

```
ENDSELECT
```

```
CHANGE FIELD(#DEPARTMENT) TO(FLT)
```

```
ENDROUTINE
```

EVTROUTINE HANDLING(#PHBN_PrintEmplistD.Click)

EXCHANGE FIELDS(#DEPARTMENT #PRN_NAME) OPTION(*ALWAYS)

USE BUILTIN(CALL_SERVER_FUNCTION) WITH_ARGS(MYSERVER

ENDROUTINE

END_COM

** SERVER SIDE FUNCTION

FUNCTION OPTIONS(*DIRECT) RCV_LIST(#PRNLIST)

DEF_LIST NAME(#PRNLIST) FIELDS(#PRN_NAME #PRN_LOC) TYPE(

USE BUILTIN(LIST_PRINTERS) WITH_ARGS(A) TO_GET(#PRNLIST #S

WPPS (Windows Printing Setup File)

The Windows Printing Setup File specifies the file, including the full path, which contains the associated printer settings. These settings will be used to initialize the printer dialog. If the printer dialog is not displayed these settings will be automatically sent to the printer. The default printer settings on the PC will be used if WPPS is not specified. If WPPS is specified the default printer setting will be used for any parameters not specified in the file.

When you specify the WPPS parameter, you must also specify the WPPN parameter. Even if you display the printer dialog, WPPN is required so that the dialog can be initialized with the correct default settings.

A new file can be specified at any time thus allowing the user to use different settings for different print jobs (reports, etc) without having to display the printer dialog every time.

This parameter is not passed to (inherited by) a server system. When printing from a server in a client/server environment the file name can be exchanged to the server when the print function is called or the file can specified in the server function.

The following printer settings may be specified. Some settings may not be relevant for the selected printer, in which case they will be ignored by the printer.

Note: The WPXX and PRTR parameters are X_RUN and should not be specified in this file.

PGOR= Page Orientation

Value	Description
P	Portrait
L	Landscape

DPXP= Duplex printing

Value	Description
N	Normal (non-duplex)

H	Duplex over horizontal (flip over the long edge of the page)
V	Duplex over vertical (flip over the short edge of the page)

COLR= Color printing

Value	Description
C	Colour
M	Monochrome

COPY= Number of copies

PSIZ= Paper Size

Value	Description
LETTER	Letter, 8 ½ by 11-inches
LEGAL	Legal 8 ½ by 14 inches
A4	A4 sheet, 210 x 297 millimeters
A3	A3 sheet, 297 by 420 millimeters
xxx	xxx is a numeric value defined by the Microsoft Visual C runtime which represents a specific paper size. The complete list of values may be obtained from Microsoft's MSDN library at: http://msdn.microsoft.com/en-us/library/windows/desktop/dd319099%28v=vs.85%29.aspx For example A4 paper size is named DMPAPER_A4 and has the value 9. So use PSIZ=9. Alternatively, see below for instructions on how to retrieve printer specific values.

PSRC= Paper Source

Value	Description
A	Auto
L	Lower tray
M	Middle tray
xxx	xxx is a numeric printer specific value. See below for instructions on how to retrieve printer specific values.

QLTY= Print Quality

Value	Description
H	High
M	Medium
L	Low
D	Draft
xxx	xxx is a numeric printer specific value which specifies the number of dots per inch (DPI). See below for instructions on how to retrieve printer specific values.

COLL= Collation

Value	Description
Y	Collate when printing multiple copies
N	Do not collate when printing multiple copies

PTYP= Paper Type

Value	Description
--------------	--------------------

P	Plain paper
G	Glossy paper
T	Transparent film
xxx	xxx is a numeric printer specific value. See below for instructions on how to retrieve printer specific values.

FNAM= Form Name

See below for instructions on how to retrieve printer specific values.

Printer specific values can be obtained by running the application with WPPD=E and trace settings ITRO=Y ITRL=9 and ITRM set to a value large enough to log the required tracing information. Enter the required values into the print dialog. The values returned by the print dialog will be logged in the trace file. Only values that are relevant to the current printer are logged. Open the trace file and search for the phrase "Printer Details". The following is an example of the trace output:

```
MESSAGE      : Printer Details for \\syd1\HP LaserJet 4050 Series
MESSAGE      : Printer Details... PGOR=1 DPXP=1 COLR=2 COPY=2 PSI
```

Example of a printer setup file

Example 1:

```
PGOR=P
DPXP=V
COPY=4
```

Example 2:

```
PGOR=L
DPXP=N
COLR=C
PSIZ=33
```

WPPD (Windows Printing Print Dialog)

Controls how and when users of the application are prompted to select printer details via the standard Windows printer dialog. The allowable values for this parameter are:

WPPD=D (default)

The user is not to be prompted and the default printer is to be used.

WPPD=F (first time)

The user is to be prompted the first time (within the X_RUN process) that a report is to be printed and thereafter the chosen printer used for all subsequent reports without any further prompting.

WPPD=E (every time)

The user is to be prompted every time (within the X_RUN process) that a report is to be printed.

WPPD=A (automatic)

The user is to be prompted the first time (within the X_RUN process) that a report is to be printed and thereafter only when a report is to be printed that has different characteristics to the last report that was printed. The report characteristics that are checked for differences are the report width, the page length, the overflow line and the last detail line.

The default value is WPPD=A.

In batch mode X_RUN commands, the parameter is always treated as WPPD=D (default) regardless of the actual value specified. This is because batch mode commands do not have access to a UI (User Interface) and thus cannot present the printer dialog to a user.

WPFD (Windows Printing Font Dialog)

Controls how and when users of the application are prompted to select font details via the standard Windows font dialog. The allowable values for this parameter are:

WPFD=D (default)

The user is not to be prompted and the default font is to be used.

WPFD=F (first time)

The user is to be prompted the first time (within the X_RUN process) that a report is to be printed and thereafter the chosen font used for all subsequent reports without any further prompting.

WPFD=E (every time)

The user is to be prompted every time (within the X_RUN process) that a report is to be printed.

WPFD=A (automatic)

The user is to be prompted the first time (within the X_RUN process) that a report is to be printed and thereafter only when a report is to be printed that has different characteristics to the last report that was printed. The report characteristics that are checked for differences are the report width, page length, overflow line and the last detail line.

The default value is WPFD=A.

In batch mode X_RUN commands the parameter is always treated as WPFD=D (default) regardless of the actual value specified. This is because batch mode commands do not have access to a UI (User Interface) and thus cannot present the font dialog to a user.

WPDF (Windows Printing Default Font)

Specifies the default font to be used.

Where a font name contains imbedded blanks use double quotes to specify the name (for example, WPDF="MS LineDraw").

The default value "Courier New" is used, if this parameter is not provided.

WPDS (Windows Printing Default Font Size)

Specifies the default font point size to be used.

The default value is 8.

WPFO (Windows Printing Fixed Pitch Only)

WPFO=Y or WPFO=y indicates that when a user is to be prompted for fonts via the standard font dialog that only fixed pitch fonts be shown for selection.

Any other value indicates that all valid fonts for the selected printer should be shown.

The default value is WPFO=N (i.e. all valid fonts for the printer are shown).

WPAS (Windows Printing Automatic Stretching)

WPAS=Y or WPAS=y indicates that the automatic stretching of report pages should be enabled.

Any other value indicates that automatic stretching is not enabled.

The default value is WPAS=N.

When enabled, automatic page stretching causes a watch to made for the longest page that was printed in a report. This page is then logically stretched so that the entire printed page is covered by it. All other pages in the report are proportionately stretched the same way.

Automatic stretching is only enabled for a report when WPAS=Y is specified and the printed report contains at least 2 pages.

Questions and Answers

When I submit a batch job what extended printing options does it inherit ?

When you submit a batch job or start a remote batch job by connecting to a server system the current settings for all WPxx= parameters are inherited by it. However, you should note the following:

- Parameter values WPPD=D and WPF D=D are always assumed in batch jobs because batch jobs cannot communicate with the user via any user interface.
- The WPDF= and WPDS= values used are the ones selected for the last report produced by the submitting job.
- The default printer for a user on a server may be different to their default printer on a client system.

Can I choose Portrait or Landscape mode?

Yes. These are options in the Windows printer dialog.

Why are the defaults wrong when the printer dialog appears?

When the printer dialog appears it uses the Windows environmental default settings for the selected printer (eg: the paper size). If these defaults are wrong then alter them via the standard Windows facilities.

What type and size of fonts should I use?

If your report has lots of inserted text (rather than just fields, labels and column headings) then you should look to use fixed pitch fonts such as MS LineDraw, Courier or Lucida Console.

For reports that do not contain a lot of inserted text, variable pitch fonts such as Arial or Times New Roman may work very well.

For 80 wide reports printed in portrait mode font sizes of 8 through 11 should be used. A common problem for 80 column reports is often in the report length rather than the report width. For example a 12 point font may work well with 80 columns in terms of width, but 66 lines of length may cause an overlapping of printed lines because 66 lines of a 12 point font cannot be squeezed into the vertical space available.

For 132+ wide reports printed in landscape mode, font sizes of 7 or 8 should be used.

Why is there a lot of blank space on the bottom of my reports?

This is because LANSAs reports have been configured for IBM i line printer environments. Typically there are 66 lines x 132 columns with a last print line of 60. Footings are often printed on line 57 (or less).

This means that there may be up to 9 unused lines on the end of your report. While this works well when printing on line printer stationery with its wide margin physically at the bottom (or top) it often looks wasteful when printed on A4 paper.

To decrease the number of unused lines on a report page, simply increase the OVERFLOW parameter value to a higher number such as 65 or 66.

Alternatively, on multiple page reports, try using the WPAS (automatic stretching) parameter so that the longest page printed is logically stretched to cover the entire printed page, and all other report pages are proportionately stretched.

17.4.18 XCMD Parameter

XCMD obscures the command line. Y (Yes) or N (No).

When you include this parameter, the values of a subset of parameters is replaced with asterisks (*) so that viewing the properties will not show parameters such as the password.

For example:

```
C:\PROGRAM
FILES\LANSA\X_WIN95\X_LANSA\EXECUTE\X_RUN.EXE
PROC=MYPROC LANG=ENG PART=DEX USER=**** DBUS=***
PSWD=*** DBII=LXDEVPGM DBIT=SQLANYWHERE PRTR=LPT1
DEBUG=N ITRO=N ITRM=20000 ITRL=4 ITRC=ALL ITHP=X
LOCK=YES XCMD=Y
```

The parameters for which *** are inserted are: USER, GUSR, DBUS, PSWD, PSUS, ASUS, PSPW and ASPW.

Note: Passwords (PSWD, PSPW and ASPW) are always obscured regardless of this parameter's setting.

Default is N.

If XCMD=Y, the CONNECT_SERVER Built-In Function clears the X_RUN exceptional and Server exceptional arguments. If you use XCMD=Y then you should ensure that your code calls DISCONNECT_SERVER, then DEFINE_ANY_SERVER before calling CONNECT_SERVER again.

17.5 Permanently Specify X_RUN Parameters

Parameters used with X_RUN can be specified in three ways:

- As command line parameters described in [17.1 Using X_START as a Front End to X_RUN](#).
- Via the environment variable X_RUN, described in [17.5.2 Using an Environment Variable](#).
- Via the profile file x_lansa.pro as described in [17.5.3 Using an X_LANSA.PRO Profile File](#).

The order of precedence for these options is:

1. The command line.
2. The X_RUN environment variable.
3. The x_lansa.pro profile file.

The order of precedence for a parameter specified more than once in the same place is the last value processed.

Also see

[17.5.1 Why not put your X_RUN Commands behind Icons?](#)

17.5.1 Why not put your X_RUN Commands behind Icons?

You can create multiple icons for your user objects, rather than use a menu.

For example, let's say you have a process that has a customer, order, and a product function attached to it. Instead of having the user execute the process and then select one of three functions, you could create 3 separate icons:

- Customer
- Order, and
- Product.

All three functions could be executed at once and the user would not have to cancel one function to work with another.

17.5.2 Using an Environment Variable

You can set system wide values for X_RUN parameters by setting up an environment variable called X_RUN.

For example, if you put this line into your CONFIG.SYS file:

```
SET X_RUN=LANG:ENG
```

you have indicated that unless specified on a command line (because it has a higher precedence), the LANG parameter should be English.

Similarly:

```
SET X_RUN=PRTR:LPT3 LANG:FRA
```

or, for Linux:

```
X_RUN="PRTR:lpt3 LANG:FRA"; export X_RUN
```

sets up all X_RUN commands to have printer LPT3 and to use French (unless overridden by specific command line parameters).

Note: A colon (":") is used instead of the equal sign ("="). The parameter is specified as PRTR:LPT3 rather than PRTR=LPT3.

You must use the ":" format when using the SET command.

You may use either "=" or ":" format in the command line and profile file.

17.5.3 Using an X_LANSA.PRO Profile File

Most X_RUN parameter values can be permanently specified in a special profile file named x_lansa.pro.

A profile file can provide you with three benefits:

1. No need to type in most of the parameters every time you wish to use the X_RUN command.
2. All parameter values are consistently specified. Changing and forgetting parameters between different executions of X_RUN may produce differing results that confuse you.
3. Values for parameters that are not set up when you execute your application directly from within Visual LANSA can be specified.

The partition parameters (PART=) and drive parameters (DRIV=), if not the defaults, must be specified on the command line to indicate the location of the profile file.

This file can be created and edited by most standard source editors. Once created, the x_lansa.pro file can be edited in Visual LANSA from System Information or the Remote System associated with the other system.

You must observe these rules:

- If the first character of a line is a ";" then the line is ignored.
- Lines must be less than 256 characters long.
- As many parameters as can fit in 255 characters may be specified on a single line.
- Multiple parameters, specified on a single line, must be separated by space (i.e. blank) characters.
- Details can be encrypted using the Visual LANSA interface to the file but once encrypted they cannot be modified.

For example:

Always use a LAN drive for the RRN files and English:

```
RPTH=S:\RRNDIR\ LANG=ENG
```

If you directly specify an RPTH= parameter with your X_RUN command, then x_lansa.pro should exist in the named RPTH= directory.

If you do not directly specify an RPTH= parameter on your X_RUN command,

then x_lansa.pro should exist in the x_lansa\x_ppp partition directory (where "ppp" is the partition identifier) in the path containing Visual LANSa (which is specified by DRIV= or PATH=).

It is recommended that you do not ever directly specify an RPTH= parameter on your X_RUN command. If you need to specify an RPTH= parameter, create an x_lansa.pro profile file in the x_lansa\x_ppp partition directory and include the RPTH= parameter value into it. This way you will not forget to include the correct RPTH= parameter.

If the RPTH= parameter is not included, or if its value is changed from session to session, the file relative record numbers may be assigned from different areas, producing strange results.

17.6 Database Connections

LANSA and SQL Server - Configuration Options

The Visual LANSA database connection has been enhanced to support windows authentication for Oracle and SQL Anywhere as well as continuing to support Microsoft SQL Server windows authentication.

The database connections also support the User ID and password being specified in the ODBC DSN, if supported by the database so that they are not stored in the Windows registry by Visual LANSA.

The default connection is to use the existing connection information stored in the Windows registry by Visual LANSA. This establishes a connection exactly as previously and is for backward compatibility.

The specific connections attempted by the Visual LANSA for each database type is as follows:

SQL Anywhere

1. Attempt a trusted connection
2. Connect just using the DSN. This presumes that the ODBC DSN contains all the connection information required.
3. Attempt to connect using DBA/SQL. This will be the most common connection made with new installations.
4. If connection to load Other Files, prompt for more connection information, otherwise display an error that ODBC DSN needs to be re-configured and provide an option to start the ODBC Administrator. Either connection is successful or Visual LANSA exits.

Microsoft SQL Server

1. Connect just using the DSN. This presumes that the ODBC DSN contains all the connection information required.
2. Prompt for more connection information. Either connection is successful or Visual LANSA exits.

Oracle

1. Attempt a trusted connection
2. Connect just using the DSN. This presumes that the ODBC DSN contains all the connection information required.

3. Attempt to connect using DBA/SQL. This will be the most common connection made with new installations.
4. Prompt for more connection information. Either connection is successful or Visual LANSA exits

Refer also to [LANSA and SQL Server - Configuration Options](#) in the Tips and Techniques on the LANSA web site.

17.7 The .XQ* Files

17.7.1 Tips for Setting up and Using .XQ* Files

Any PC that executes an application using DBID=*ANY, DBID=*AS400, DBID=*OTHER, DBID=*NONE or DBII=*NONE needs to have a special set of additional files created or installed on it, or available to it via a connected server disk drive.

These files are called the "xq*" files and are accessed by your generated Visual LANSA application(s) when users perform actions such as prompting, display help text, etc.

Values *NONE, *ANY, *AS400 and *OTHER are identical in indicating that no local DBMS system is to be used. The difference is that the value *ANY, *AS400 or *OTHER is used to additionally indicate that an automatic connection to a Server is to be established. The details of the connection are defined by a series of PSXX= parameters which are described in [17.4.12 PSxx Server Parameters](#).

The .XQ files are used in place of a local DBMS for read-only dictionary/repository access.

The complete set of these files is:

Name	Description
lx_msg.xqi	Message definitions
lx_msg.xqd	
lx_f96.xqf	System definition (not used on IBM i)
lx_f46.xqf	Partition "ppp" definition
lx_f60.xqf	Partition "ppp" languages
lx_f03.xqi	Partition "ppp" field definitions
lx_f03.xqd	
lx_f40.xqi	Partition "ppp" program defined and overridden field definitions
lx_f40.xqd	
lx_f62.xqi	Partition "ppp" field descriptions

lx_f62.xqd
lx_f04.xqi Help text
lx_f04.xqd
lx_f61.xqi *MTXT values
lx_f61.xqd
lx_f05.xqi Validation rule definitions
lx_f05.xqd
lx_f06.xqi Validation rule definitions
lx_f06.xqd
lx_f07.xqi Validation rule definitions
lx_f07.xqd
lx_f08.xqi Validation rule definitions
lx_f08.xqd
lx_f09.xqi Validation rule definitions
lx_f09.xqd
lx_f10.xqi Validation rule definitions
lx_f10.xqd
lx_f11.xqi Validation rule definitions
lx_f11.xqd
lx_f44.xqi Process attachments
lx_f44.xqd
lx_f64.xqi Physical file descriptions
lx_f64.xqd
lx_f65.xqi Logical file descriptions
lx_f65.xqd

lx_f66.xqi Process descriptions
lx_f66.xqd
lx_f67.xqi Function descriptions
lx_f67.xqd
lx_f27.xqi File Definitions
lx_f27.xqd
lx_f15.xqi Logical Files
lx_f15.xqd
lx_f14.xqi File Fields
lx_f14.xqd
lx_f26.xqi System Variables
lx_f26.xqd

These files are in binary format. They are "read only" files and should not be manually edited with any type of tool.

Failure to observe this rule may lead to application failure and/or unpredictable results.

When created, each set of lx_fnn.xqi and lx_fnn.xqd are a "matched pair". File lx_fnn.xqi is an index and makes direct offset references into the lx_fnn.xqd (data) file. You must always create, ship and install both of the lx_fnn files together as a pair. Failing to observe this rule may lead to application failure and/or unpredictable results.

- The files lx_f61.xqd and lx_f61.xqi are only required if your application uses *MTXT variables.
- The files lx_f05.xqd through to lx_f11.xqi are only required if you use (or have enabled) automatic field level help text generation.
- The files lx_f27.xqd through to lx_f26.xqi are only required for LANSAs Open applications (on the Server).
- The files lx_f96.xqf through lx_f60.xqf are only required for applications that

require non-default system, partition, and language definitions. File lx_f96.xqf is not used for LANSA for i.

- If connecting to a Server via the PSXX parameters or an INIT function, repository data that is not found locally can be retrieved from the Server, or set to defaults. Theoretically, this means that you can deploy the client side of a SuperServer application with no xq* files. However, there are performance and tailoring setbacks. Refer to [PSRR Notes \(Primary Server Route Repository\)](#) for more details.

Refer to [Client/Server Applications](#) in the *LANSA Application Design Guide* for more details about designing and building client/server applications.

All lx_fnn files are placed into the partition source directory (x_lansa\x_ppp\source). If a file does not exist it is created. If it does, it is cleared of all existing data before the export proceeds.

In execution environments, when a specific file is being read, the x_lansa\x_ppp\source directory is always checked first. If the file cannot be located in that directory, the x_lansa\source directory is checked. This allows flexibility in deployment. For example, different partitions can have their own copy of the message file.

The xq* files may be created on any Windows development system at any time. To do this use the X_RUN command like this:

X_RUN PROC=*SYSEXPORT FUNC=tttt LANG=xxx PART=ppp

The "tttt" value can be specified as an individual table name (e.g. LX_F03 or LX_F46) or as *ALL. The special value *ALL causes all the output files to be produced.

Once started, the export process displays a standard event log and updates it as the export proceeds. When complete, an "OK" button will appear in the event log window, allowing you to review the messages. When you click on the "OK" button the export process ends.

- The PROC=*SYSEXPORT facility exports the specified file(s) to all enabled environments.
- Where an applicable code page / environment translation file exists, its presence and use will be noted in the messages issued by the *SYSEXPORT facility.
- You cannot use PROC=*SYSEXPORT with DBID=*NONE/*ANY/*AS400/*OTHER. Failure to observe this rule may

lead to unpredictable results and/or application failure.

- You cannot use PROC=*SYSEXPORT when the lx_fnn files are in use by another application or user. Failure to observe this rule may lead to application failure and/or unpredictable results.
- For lx_f96 and lx_msg, which are system based, all details are exported. For all other lx_fnn files, only the details from the current partition (i.e. PART=ppp on the X_RUN command) are included.
- The partition based lx_fnn files do not store the partition internally. Therefore they can be used with a different partition name. However, data library and program library values are still stored within the files, and these cannot be changed.
- Normally lx_fnn files are created at the completion of a task / project and included into the packaging and installation procedures of the application. If you are using lx_fnn files and DBID=*NONE/*ANY/*AS400/*OTHER, then you should add their production, shipping and installation procedures to the procedures recommended in the section in this guide that describes the packaging of applications.
- The lx_fnn files are a read-only "snapshot" of the associated SQL based tables. You must remember that they are a snapshot. For example, you may use *SYSEXPORT to export LX_F04 (help text) and then run your application using DBID=*NONE. If you edit the help text via Visual LANSA the change will not appear in applications using DBID=*NONE until the lx_f04.xqi and lx_f04.xqd files are (re)created again. (However, if your application connects to a Server and the repository data exists there, **missing** information will be retrieved by default, if it exists on the server. **Changed** information will **not** be retrieved. Refer to [PSRR Notes \(Primary Server Route Repository\)](#) for more details.

The lx_fnn files will support system initialization, field prompting and help text display in a totally standalone environment. These are the essential elements of being able to execute an application in a standalone environment.

Any other form of database activity will fail unless the I/O operations are "diverted" to a server.

Note that I/O operations also include object lock requests (which may be automatically diverted to the server) and IBM i data area emulations (which must be manually diverted to the server by using the CALL_SERVER_FUNCTION facility). Other Built-In Functions such as SAVE_LIST may also involve database activity and must be manually diverted

to the server.

Imagine a function that must perform some activity and give the result(s) back to the caller. For example, it might allocate the next order number, or save a list, etc.

The activity must always be performed on the server.

By structuring the function like this:

```
function *direct rcv_lsts(<working lists>)  
  
exchange <other information> options(*always)  
  
if *cputype = as400  
    <perform required activity>  
else  
    use call_server_function ( ... itself ..... )  
endif  
return
```

When called on an IBM i this function does its job and receives and returns information via working lists, exchange lists, etc.

When called in a PC application, the function immediately calls itself on the associated server (which does its job and returns the results).

To the program that called this one, the switching to the server (if required) is invisible and immaterial.

By using a more flexible and dynamic switch than "*cputype = as400" a very powerful and dynamic means of switching logic between the client and server could be designed.

User Defined Messages

User defined Messages will not be copied to LX_MSG.XQD / XQI files.

Therefore the use of FXQF=*ALL is not recommended in scenarios where you have your own application specific messages. In such cases the use of a message file is required.

17.7.1 Tips for Setting up and Using .XQ* Files

These tips and techniques may help you in setting up and using ".xq*" files:

- These files are static. They represent a "snapshot" of your dictionary/repository. For example, if you edit the help text associated with a field, then use X_RUN DBII=*NONE. The new help text will not appear until you (re)run using the X_RUN PROC=*SYSEXPORT option. Refer to [17.7 The .XQ* Files](#) for details about using X_RUN PROC=*SYSEXPORT.
- If you are packaging/bundling your application for distribution to other PCs and plan to use X_RUN DBII=*NONE on them, then you will have to include the latest ".xq*" files into your package as well. (However, if your application connects to a Server and the repository data exists there, **missing** information will be retrieved by default, if it exists on the server. **Changed** information will **not** be retrieved. Refer to [PSRR Notes \(Primary Server Route Repository\)](#) for more details.)

Only package the *.xq* files you actually need. This will save space. However, make sure that before you package your application, you test it with the subset of *.xq* files.

17.8 Lock Timeout

These notes apply to non-IBM i platforms

A lock timeout is treated by LANSAs as a fatal error unless LANSAs lock timeout handling is switched on. In fact, by default, most databases wait indefinitely for locks to be resolved and thus lock timeouts do not occur. Some databases allow a lock timeout to be set globally, for example, Oracle on Windows. If that is done and a lock timeout occurs, then a fatal error would occur with the LANSAs application.

Once you have enabled LANSAs to trap lock timeouts LANSAs will still set IO\$STS=ER. Thus if special handling is not provided by the developer, current error handling will be executed, but it will no longer be a LANSAs fatal error, its just an IO error.

Therefore, it is imperative that you check for a lock error by using the system variable *DBMS_RECORD_LOCKED. Without it, the user will not get any extra messages describing the reason for the error.

In summary, if you switch lock timeouts ON, then you will still get an IO error and you MUST check for a lock timeout by using the system variable *DBMS_RECORD_LOCKED.

Also see

[17.8.1 Lock Timeout Configuration](#)

[17.8.2 Lock Timeout Behavior Examples](#)

17.8.1 Lock Timeout Configuration

[Connection Lock Timeout](#)

[Statement Lock Timeout](#)

[SuperServer](#)

Lock Timeout Types

There are two different methods of setting a timeout for when an SQL transaction is waiting for a lock to be freed by another process. They are:

1. For a [Connection Lock Timeout](#) supported on Windows and IBM i (but not on Linux) for all LANSAs development databases: SQL Server, Oracle and Sybase Adaptive Server Anywhere, set a timeout on each connection so that ANY locks that occur on that connection can return control back to the application.
2. For Oracle on Linux, a different technique is required. In this environment, a wait time can be set on the SELECTs executed before LANSAs performs an UPDATE or DELETE. This is called a [Statement Lock Timeout](#). LANSAs also supports this setting on Oracle for Windows so that an application can expect consistent lock timeout behavior when running on either Windows or Linux.

Connection Lock Timeout

The Connection Lock Timeout requires setting LOCK_TIMEOUT in X_DBMENV.DAT to the time to wait before timing out. A value of zero indicates it should wait forever and that LANSAs should not trap timeout errors. This is for backward compatibility. Zero is the default. The unit of measurement differs depending on the database type. This is noted in comments in X_DBMENV.DAT. For example SQL Server requires the timeout to be specified in milli-seconds and MySQL requires it to be specified in seconds.

Also set LOCK_TYPE=C (default)

For Oracle it also requires setting a value in the file ORAODBC.INI in your Windows directory. This is usually c:\windows, but the actual value can be determined by typing "set windir" in a command window.

To set the lock timeout ensure that text similar to the following is in ORAODBC.INI:

[Oracle ODBC Driver Common]LockTimeOut=2

Note that for Oracle the value in X_DBMENV.DAT just enables LANSAs's lock timeout behavior, it does not actually set the timeout value. Also the error code

returned depends on which method is chosen to implement the lock so
DBMS_RETCODE_ROW_LOCKED=1013

The full set of Oracle settings in X_DBMENV.DAT is:

```
LOCK_TYPE=C
LOCK_TIMEOUT=2
CMD_LOCK_TIMEOUT=<setting ignored>
DBMS_RETCODE_ROW_LOCKED=1013
```

Statement Lock Timeout

The Statement Lock Timeout is specific to Oracle. It allows the same lock timeout behavior to occur on Windows and Linux Oracle databases.

This also requires LOCK_TIMEOUT in X_DBMENV.DAT to be set to a non-zero value to enable the LANSAs feature, but it does not actually set the value of the timeout. Also set LOCK_TYPE=S.

There is also a choice of whether to wait for the lock to be freed, or to not wait at all if a lock is encountered. This is set in CMD_LOCK_TIMEOUT: "FOR UPDATE WAIT n" or "FOR UPDATE NOWAIT" respectively.

Finally the error code (DBMS_RETCODE_ROW_LOCKED) needs to be set to match the timeout setting as set out below.

For example, to set the lock timeout to 2 seconds use the following settings:

```
LOCK_TYPE=S
LOCK_TIMEOUT=2
CMD_LOCK_TIMEOUT=FOR UPDATE WAIT 2
DBMS_RETCODE_ROW_LOCKED=30006
```

To set the lock timeout to not wait at all use the following settings:

```
LOCK_TYPE=S
LOCK_TIMEOUT=2
CMD_LOCK_TIMEOUT=FOR UPDATE NOWAIT
DBMS_RETCODE_ROW_LOCKED=54
```

SuperServer

For Windows, Linux servers, the Client and Server must have the same timeout settings in x_dbmenv.dat for the database type used on the server (LOCK_TYPE, LOCK_TIMEOUT, CMD_LOCK_TIMEOUT, and

DBMS_RETCODE_ROW_LOCKED). Otherwise,
*DBMS_RECORD_LOCKED may return N when you expect it to return Y.

17.8.2 Lock Timeout Behavior Examples

The following is a step by step description of how various combinations of LANSAs database IO work in different databases when a row is locked. These differences are fundamental to the architecture of particular database engines and thus cannot be abstracted away by LANSAs to provide consistent behavior across all database engines.

This is not an exhaustive list of database behaviors. It just seeks to show how the database engines may differ and thus not to include presumptions about that behavior if cross-database consistency is required.

It also shows that thorough testing is required when deciding to use a new database engine with an application. LANSAs removes many of the concerns of running an application against different databases, but there are still some subtle differences that cannot be assuaged by LANSAs.

See test results for:

[Adaptive Server Anywhere 9.0](#)

[SQL Server 2005](#)

[Oracle 10.2 - Connection Lock](#)

[Oracle 10.2 - Statement Lock](#)

[PC Other Files](#)

Adaptive Server Anywhere 9.0

ODBI=2 (READ_COMMITTED)LockTimeout=2 (X_DBMENV.DAT)

WITH_KEY IO access

Step	Action (User)	IO\$STS	Message
1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	VE	Record already exists
3	Delete (2)	ER	Record Locked
4	Commit (1)		
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	ER	Record Locked
8	Fetch (2)	ER	Record Locked
9	Commit (1)		
10	Delete (1)	OK	IO Operation Succeeded
11	Insert (2)	ER	Record Locked
12	Update (2), Delete (2)	NR	Record not found
13	Commit (1)		

Note that ASA is behaving as if it is seeing the uncommitted actions of the other user, even though the Transaction Isolation Level is set to READ_COMMITTED.

Last Record Read IO access

This is a different set of steps as there must be a record to read before the last record read can be updated or deleted! Hence when an error occurs, the record must be fetched again like in Step 13.

Step	Action (User)	IO\$STS	Message
------	---------------	---------	---------

1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	VE	Record already exists
3	Commit (1)		
4	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	ER	Record Locked
8	Fetch (2)	ER	Record Locked
9	Commit (1)		
10	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
11	Delete (1)	OK	IO Operation Succeeded
12	Insert (2)	ER	Record Locked
13	Fetch (2) Update (2), Fetch (2) Delete (2)	NR	Record not found
14	Commit (1)		

Note in Step 13 that the Update and Delete cannot proceed because the record is not found.

SQL Server 2005

ODBI=2 (READ_COMMITTED)LockTimeout=2 (X_DBMENV.DAT)

WITH_KEY IO access

Step	Action (User)	IO\$STS	Message
1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	ER	Record Locked
3	Delete (2)	ER	Record Locked
4	Commit (1)		
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	OK	IO Operation Succeeded
8	Fetch (2)	ER	Record Locked
9	Commit (1)		
10	Delete (1)	OK	IO Operation Succeeded
11	Insert (2)	ER	Record Locked
12	Update (2), Delete (2)	ER	Record Locked
13	Commit (1)		

Note the differences between ASA and SQL Server at steps 2, 7 and 12. There may be many ramifications of these differences in attempting to have an application perform the same way on both databases. For example, Step 7 implies that if your application updates a record and doesn't commit it, then re-reads the record, on SQL Server it will work. When you then execute it on ASA, it will timeout if a LockTimeout is set, otherwise it will block.

Last Record Read IO access

This is a different set of steps as there must be a record to read before the last

record read can be updated or deleted! Hence when an error occurs, the record must be fetched again like in Step 13.

Step	Action (User)	IO\$STS	Message
1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	ER	Record Locked
3	Commit (1)		
4	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	OK	IO Operation Succeeded
8	Fetch (2)	ER	Record Locked
9	Commit (1)		
10	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
11	Delete (1)	OK	IO Operation Succeeded
12	Insert (2)	ER	Record Locked
13	Fetch (2) Update (2), Fetch (2) Delete (2)	ER	Record Locked
14	Commit (1)		

Note that step 13 gets the lock timeout on the Fetch not the Update. Compare this with ASA which reports the row does not exist on the Fetch and Oracle which succeeds on the Fetch but gets the lock timeout on the Update.

Oracle 10.2 - Connection Lock

ODBI=2 (READ_COMMITTED)
LockTimeout=2 (ORAODBC.INI)

WITH_KEY IO access

Step	Action (User)	IO\$STS	Message
1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	ER	Record Locked
3	Delete (2)	NR	Record not found
4	Commit (1)		
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	OK	IO Operation Succeeded
8	Fetch (2)	OK	IO Operation Succeeded
9	Commit (1)		
10	Delete (1)	OK	IO Operation Succeeded
11	Insert (2)	ER	Record Locked
12	Update (2), Delete (2)	ER	Record Locked
13	Commit (1)		

Note the differences between Oracle and SQL Server at steps 3 and 8.

Last Record Read IO access

This is a different set of steps as there must be a record to read before the last record read can be updated or deleted! Hence when an error occurs, the record must be fetched again like in Step 13.

Step	Action (User)	IO\$STS	Message

1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	ER	Record Locked
3	Commit (1)		
4	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	OK	IO Operation Succeeded
8	Fetch (2)	OK	IO Operation Succeeded
9	Commit (1)		
10	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
11	Delete (1)	OK	IO Operation Succeeded
12	Insert (2)	ER	Record Locked
13	Fetch (2) Update (2), Fetch (2) Delete (2)	ER	Record Locked
14	Commit (1)		

Oracle 10.2 - Statement Lock

**ODBI=2 (READ_COMMITTED)
FOR UPDATE WAIT n (X_DBMENV.DAT)**

The lock timeout is only set for SELECT operations, and then only for reads before UPDATE and DELETE. Its not possible to put the lock timeout on other SELECTs because a lock is applied too which is undesirable for all SELECTS. Only specific ones should be locked. INSERTS do not provide a WAIT option and thus wait for the row to be unlocked (block).

WITH_KEY IO access

Step	Action (User)	IO\$STS	Message
1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	Block	
3	Delete (2)	NR	Record not found
4	Commit (1)		
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	OK	IO Operation Succeeded
8	Fetch (2)	OK	IO Operation Succeeded
9	Commit (1)		
10	Delete (1)	OK	IO Operation Succeeded
11	Insert (2)	Block	
12	Update (2), Delete (2)	ER	Record Locked
13	Commit (1)		

Note the highlighted lines 2 and 11 differ from Lock Timeout results for Oracle. For the inserts, it's not possible to set a timeout, so they block until the transaction is committed.

Last Record Read IO access

Same results as for WITH_KEY in terms of how using FOR UPDATE differs from a LockTimeout.

Step	Action (User)	IO\$STS	Message
1	Insert (1)	OK	IO Operation Succeeded
2	Insert (2)	Block	
3	Commit (1)		
4	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
5	Update (1)	OK	IO Operation Succeeded
6	Update (2)	ER	Record Locked
7	Fetch (1)	OK	IO Operation Succeeded
8	Fetch (2)	OK	IO Operation Succeeded
9	Commit (1)		
10	Fetch (1), Fetch (2)	OK	IO Operation Succeeded
11	Delete (1)	OK	IO Operation Succeeded
12	Insert (2)	Block	
13	Fetch (2) Update (2), Fetch (2) Delete (2)	ER	Record Locked
14	Commit (1)		

The conclusion drawn from this testing is that if Oracle is being used on Linux and Windows then use the FOR UPDATE option on BOTH platforms. Otherwise use a LockTimeout. And even so, if different databases are used, be very wary that behavior is different for different databases and some database specific code may be required in some situations.

PC Other Files

The lock timeout is separately set per database type. Therefore if a lock timeout is set, say, for Adaptive Server Anywhere, it will be set on all PC Other Files that use Adaptive Server Anywhere.

Full set of tests do not require testing every database with every other database. This set of tests should be sufficient:

	Main DB	Other DB	Other DB	Other DB
Test Set 1	ASA	SQL Server	Oracle (C)	MS Access
Test Set 2	SQL Server	ASA	MySQL	
Test Set 3	Oracle (C)			
Test Set 4	Oracle (S)	SQL Server		

The first 3 tests are to use LOCK_TYPE=C for all databases.

Test Set 4 should use LOCK_TYPE=S for Oracle and of course LOCK_TYPE=C for SQL Server.

17.8.3 Technical Implementation Details

The LANSAs Database Layer will use LOCK_TYPE=C to set a lock timeout for all databases except Oracle – Oracle requires the timeout to be set in ORODBC.INI. Soon after the connection is created to the database, the command setting in CMD_LOCK_TIMEOUT will be executed on the connection.

For LOCK_TYPE=S, all databases will append the value in CMD_LOCK_TIMEOUT to the selects performed before UPDATE and DELETE. These statements are flagged by the OAM using **pX_Ids->chOAMRqsType |= X_LOCK_OPERATION**. This has only been tested with Oracle so far.

17.8.4 The X_DBMENV.DAT File

The x_dbmenv.dat file defines the characteristics of the various DBMS (Database Management Systems) that your Visual LANSA applications may use.

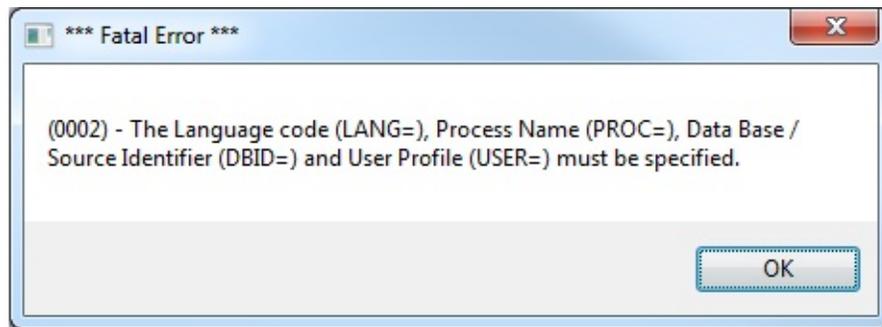
This file is shipped with your installation media and installed automatically. You can view the file in text format by selecting the *Settings and Administration* in the main LANSA program group and selecting *Databases* from the list.

These X_DBMENV.DAT file parameters: LOCK_TYPE, LOCK_TIMEOUT, CMD_LOCK_TIMEOUT and DBMS_RETCODE_ROW_LOCKED are described in detail in [Lock Timeout](#).

The contents of this file should only ever be changed if you receive instructions, in writing, from your LANSA product vendor.

17.9 User Instructions for Microsoft Exception or Dr Watson

All these error types are generically called exceptions. When an exception occurs, it is not due to a program error on your part. LANSAs specifically traps all errors that you may cause by displaying a Fatal Error message box with a number as in the following example:

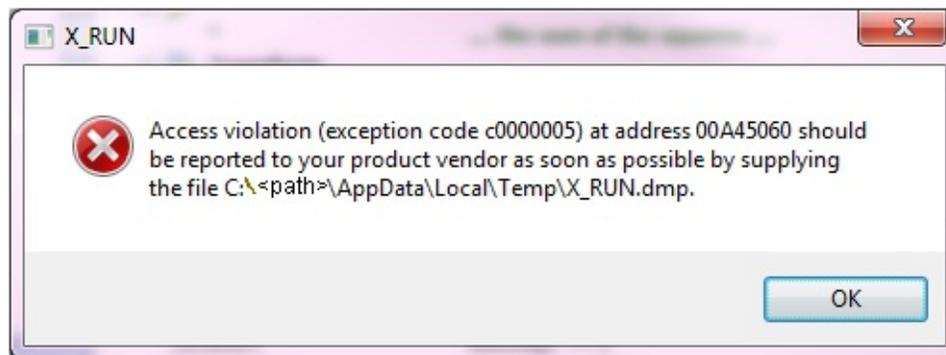


If you experience an exception during the execution of your application (similar to the example below), you should follow the steps below to gather the maximum amount of information for reporting this error to your LANSAs product vendor. Providing as much information as possible when reporting the issue can significantly increase the resolution of the problem. Failure to carry out these steps will probably delay the resolution of the issue. Armed with the requested information, your Product Vendor may be able to identify a workaround overnight.

Note that instructions on how to set the ITRO and ITHP command line parameters in a deployed environment can be found in this guide by searching for the parameter name. In a development environment, these options are asked for when executing a Process, Function or Form.

1. Write down the address provided in the message.
2. Clean up any existing trace files by searching for and deleting all `x_trace*` files from `\x_win95\x_lansa`. DO NOT DELETE `X_ERR.LOG`.
3. Turn tracing on (`ITRO=Y`). Re-run the application and attempt to reproduce the exception. This will produce an `x_tracennn.txt` trace file in the `x_win95\x_lansa` folder (where `nnn` can be 001, 002, 003 etc.).
4. Turn tracing on (`ITRO=Y`) and Pointer Validation (`ITHP=P`). Re-run the application and attempt to reproduce the exception. This step will not overwrite the previous `x_trace*` file(s) but will create a new `x_tracennn.txt` file.

5. If step 3 does not reproduce the problem, restrict the trace to just heap messages (ITRC=HEP). Re-run the application and attempt to reproduce the exception. This step will not overwrite the previous x_trace* file(s) but will create a new x_tracennn.txt file.
6. Turn tracing on (ITRO=Y) and validation of all heaps (ITHP=A). Re-run the application and attempt to reproduce the exception. This step will not overwrite the previous x_trace* file(s) but will create a new x_tracennn.txt file.
7. Send ALL the x_trace* files in \x_win95\x_lansa to ensure that your product vendor receives the correct ones. Please indicate in which of the above steps you were able to reproduce the exception.
8. Also send the dump file and the x_err.log file which are in the directory displayed in the fatal error message. In the sample error below, the x_err.log file is located on c:\<path>\AppData\Local\Temp. The location of the x_err.log file varies from machine to machine and user to user. If you do not find x_err.log in this directory, search your disk drives for the x_err.log file.



The title of the message will reflect the Windows executable that you are currently executing. Other examples are "LANSA" and "X_DLL".

9. Report the LANSA version and EPCs that you have installed. This is as critical as the traces.
In the development environment, the LANSA version and EPC level can be obtained from the Product Information menu item in the Help menu. In the deployment environment, you can obtain the build number from \x_win95\x_pkgs\x_boot\nnnn. Report the "nnnn" part of the directory name, e.g. 2208. To determine the EPCs, navigate into this directory, if there is an EPC directory here, go into that. The EPCs will be listed as directory names: e.g. EPC616. Report the EPC numbers or None if there is not an EPC directory. If you do not have an X_BOOT directory, ask your administrator or

the product vendor for the version and EPC information.

18. Error Messages

You will link directly to these error codes provided you have access to the relevant online guide:

- [LANSA Open Error Codes](#)
- [Error Code 23 - CPI-C Return Codes](#) (Host Integration Server 2000, Network Services for DOS).
- [Communications Error Codes](#) in the *LANSA Communications Setup Guide*.
- [Escape Error Message](#) in the *LANSA for i User Guide*.

Appendix A. Other_Vendor Built-In Functions

Available Built-In Functions:	Id	DLL	Description
OV_FILE_DIALOG	997	U_BIF997	Presents standard File dialog.
OV_FILE_SERVICE	992	U_BIF992	Requests a file/directory service.
OV_INDEXED_SPACE	989	U_BIF989	Creates and manages an indexed space.
OV_MESSAGE_BOX	998	U_BIF998	Presents a message box.
OV_PASTE_CLIPBOARD	996	U_BIF996	Pastes information from the clipboard.
OV_POST_CLIPBOARD	995	U_BIF995	Posts information to the clipboard.
OV_QUERY_SYS_INFO	993	U_BIF993	Query LANSAs system information.
OV_SLEEP	986	U_BIF986	Pauses the program for milliseconds.
OV_SOUND_ALARM	999	U_BIF999	Sounds an alarm.
OV_SYSTEM_SERVICE	991	U_BIF991	Requests a system service.

Disclaimer

These Built-In functions are provided as working examples and demonstrations for your use (with all source code). No warranty is expressed or implied in this provision.

Your attention is drawn to the following disclaimer that is included in the source code of all other vendor and user defined Built-In functions :

```
/* =====  
/* ===== USER DEFINED BUILTIN FUNCTION DEFINITION =
```

```

/* =====
/*
/*                                     */
/* This is a sample of how a user defined builtin function may be */
/* defined. It is provided as an example only. No warranty of any */
/* kind is expressed or implied. The programmer copying this code */
/* is responsible for the implementation and maintenance of this */
/* function, both initially and at all times in the future.      */
/*                                     */
/* User defined builtin functions are a powerfull facility. However, */
/* you should note that YOU are responsible for any impact the */
/* use of a user defined builtin function has on the performance, */
/* security, integrity, portability and maintainability of your */
/* applications.                                               */
/*                                     */
/* N.B. In general, do not use MessageBox api or any other */
/* operating system-specific api that requires user input if the BIF */
/* is expected to work in Server environments, like LANSA for Web. */
/* There is a version of MessageBox available within LANSA that DOES */
/* work in server environments - X_PDF_PromptYesNoCancel & */
/* X_PDF_PromptYesNo. See x_pdfpro.h for prototypes */
/*                                     */
/* Note that MessageBox has been allowed in User Defined BIFs, unlike*/
/* the rest of LANSA, so it is up to you, the developer, to ensure */
/* any use of it is valid. (x_glodef.h automatically allows it if */
/* U_BIF_FUNCTION is defined). */
/* =====

```

Support, Questions?

Any problems you have with these Built-In Functions, or questions that you have about their use, can be directed through your LANSA distributor who will pass the details directly to LANSA Support. Please indicate clearly that the problem or question relates to an "Other Vendor" Built-In function so that LANSA support can pass the details on to the vendor who supplied the Built-In function.

OV_FILE_DIALOG

⇒ **Note:** Built-In Function Rules

Select a fully qualified file name from a file dialog.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 997

DLL Required U_BIF997.DLL

:

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Dialog title. Default is "File".	1	25		
2	A	Opt	Initial path, file name or filter. Default is "*.*"	1	255		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Fully qualified file name.	1	255		
2	A	Req	Return value indicating if a file was successfully selected. OK = selection was successful ER = selection was cancelled	2	2		

OV_FILE_SERVICE

⇒ **Note:** Built-In Function Rules

Performs basic file and directory services.

Note: DO NOT ALTER this OV Built-In Function as it is used by LANSAs programs. If you create a customized version of this Built-In Function, create a copy and amend the copy.

All Windows path names support environment variable substitution.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 992

DLL Required: U_BIF992.DLL

For use with

Visual LANSAs for Windows	YES	
Visual LANSAs for Linux	YES	
LANSAs for i	Yes	Only available for RDMLX.

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Type of file service required. Pass as one of: MAKE_DIR Make Directory. REMOVE_DIR Remove Directory. REMOVE_DIR_TREE Remove a directory tree recursively. Be	1	256		

			<p>careful!</p> <p>CHECK_DIR Check if directory exists.</p> <p>CHECK_FILE Check if a file exists.</p> <p>SET_FILE Set a file's attribute to read only or normal (read/write).</p> <p>COPY_FILE Copy a file to another file.</p> <p>REMOVE_FILE Remove file.</p> <p>GET_DIR Get contents of a directory.</p> <p>COPY_DIR Copy a directory and all its sub-directories to another directory. Any matching files in the target directory will be replaced.</p> <p>COPY_PATTERN Copy files matching the specified pattern to another directory. Any matching files in the target directory will be replaced.</p>				
2	A	Opt	<p>Requested Service Argument 1</p> <p>When Arg 1 is Pass this argument as:</p> <p>MAKE_DIR Name of directory to be made.</p> <p>REMOVE_DIR Name of directory to be removed.</p> <p>REMOVE_DIR_TREE Name of the directory to be removed.</p> <p>CHECK_DIR Name of directory to be checked for.</p>	1	256		

			<p>CHECK_FILE Name of file to be checked for.</p> <p>SET_FILE Name of file to be set.</p> <p>COPY_FILE Name of file to be copied from.</p> <p>REMOVE_FILE Name of file to be removed /deleted.</p> <p>GET_DIR Name of directory whose contents are to be returned.</p> <p>COPY_DIR Name of the directory to be copied.</p> <p>COPY_PATTERN Fully qualified path with file pattern to be copied. Note * is the only wildcard that is supported.</p>				
3	A	Opt	<p>Requested Service Argument 2.</p> <p>When Arg 1 is Pass this argument as:</p> <p>MAKE_DIR Not required. Do not pass.</p> <p>REMOVE_DIR Not required. Do not pass.</p> <p>REMOVE_DIR_TREE Optional (only supported on MS Windows and Linux). Pass FORCE to delete every file even if a files is READ-ONLY. Any other value, or no value, will return an error if a file is read-only.</p> <p>CHECK_DIR Not required. Do not pass.</p>	1	256		

		<p>CHECK_FILE Not required. Do not pass.</p> <p>SET_FILE Pass as READ_ONLY or NORMAL.</p> <p>COPY_FILE Name of file to be copied to.</p> <p>REMOVE_FILE Optional. (only supported on 32 bit MS Windows) Pass FORCE to delete every file even if a file is READ-ONLY. Any other value, or no value, will return an error if a file is read-only.</p> <p>GET_DIR Optional file suffix to select files of only a specific type when retrieving the contents of a directory (e.g: DLL, EXE, DOC). Do not pass or pass as blanks to select all files.</p> <p>COPY_DIR Name of the directory to be copied to.</p> <p>COPY_PATTERN Name of the directory to be copied to.</p>				
--	--	---	--	--	--	--

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Basic Return Code. OK = Completed normally. ER = Error occurred.	2	2		
2	N	Opt	Extended Error Code. This is the operating	1	15	0	0

			system error code (when available) that may aid you in error handling or error reporting.				
3	List	Opt	<p>Returned working List. This list is only returned for certain argument 1 values as follows :</p> <p>MAKE_DIR Not returned.</p> <p>REMOVE_DIR Not returned.</p> <p>REMOVE_DIR_TREE Not returned.</p> <p>CHECK_DIR Not returned.</p> <p>CHECK_FILE Not returned.</p> <p>SET_FILE Not returned.</p> <p>COPY_FILE Not returned.</p> <p>REMOVE_FILE Not returned.</p> <p>GET_DIR The working list that is to contain the contents of the directory. It can contain from 1 to 7 fields (i.e. columns) which will be returned as the full file name. These are: the real name (full file name) the file name (without suffix), the file suffix, the file date (format YYYYMMDD), the file time (format HHMMSS), the file size (which must be a numeric field), a (sub)directory indicator (which is returned</p>				

		<p>as Y or N) and Existing contents of this working lists are cleared by this Built-In Function. Refer to the following example for more information.</p> <p>COPY_DIR Not returned.</p> <p>COPY_PATTERN Not returned.</p>				
--	--	---	--	--	--	--

Examples

The following sample RDML function (which can be copied and pasted in the CS/400 free form function editor) requests that you specify a directory name and then attempts to create it. The basic and extended return codes from the attempt to create the directory are displayed:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_DIRECT) TYPE(*CHAR) LENGTH(65);
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_ERRNO) TYPE(*DEC) LENGTH(7) DECIMALS(0) I
BEGIN_LOOP;
MESSAGE MSGTXT('Specify name of directory to be created');
REQUEST FIELDS((#OV_DIRECT *NOID));
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(MAKE_DIR #OV_DIR)
MESSAGE MSGTXT('Response from OV_FILE_SERVICE');
POP_UP FIELDS(#OV_RETC #OV_ERRNO);
END_LOOP;

```

The following sample RDML function (which can be copied and pasted in the CS/400 free form function editor) creates a directory called C:\OV_DEMO and then creates directories A, B, C and D within it. It then destroys all the directories created. Note that this is done in reverse order because a directory must be empty to be removed (destroyed):

```

FUNCTION OPTIONS(*DIRECT);
EXECUTE SUBROUTINE(DIRECT) WITH_PARMS(MAKE_DIR 'C:\OV_I
EXECUTE SUBROUTINE(DIRECT) WITH_PARMS(MAKE_DIR 'C:\OV_I

```

```

EXECUTE SUBROUTINE(DIRECT) WITH_PARMs(MAKE_DIR 'C:\OV_I
EXECUTE SUBROUTINE(DIRECT) WITH_PARMs(MAKE_DIR 'C:\OV_I
EXECUTE SUBROUTINE(DIRECT) WITH_PARMs(MAKE_DIR 'C:\OV_I
MESSAGE MSGTXT('Directories all created .... use OK to delete them now');
POP_UP FIELDS((#DATE *L3 *P2)) AT_LOC(8 23) WITH_SIZE(55 10) EX
EXECUTE SUBROUTINE(DIRECT) WITH_PARMs(REMOVE_DIR 'C:\OV
MESSAGE MSGTXT('Directories all deleted .... use OK to end this function');
POP_UP FIELDS((#DATE *L3 *P2)) AT_LOC(8 23) WITH_SIZE(55 10) EX

```

```

SUBROUTINE NAME(DIRECT) PARMs((#OV_SERV *RECEIVED) (#OV
DEFINE FIELD(#OV_SERV) TYPE(*CHAR) LENGTH(20);
DEFINE FIELD(#OV_DIRECT) TYPE(*CHAR) LENGTH(65);
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_ERRNO) TYPE(*DEC) LENGTH(7) DECIMALS(0) I
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(#OV_SERV #OV_DIRE
IF COND('#OV_RETC *NE OK');
DISPLAY FIELDS(#OV_SERV (#OV_DIRECT *NOID) #OV_RETC #OV_I
ABORT MSGTXT('Directory Operation failed');
ENDIF;
ENDROUTINE;

```

The following sample RDML function (which can be copied and pasted in the CS/400 free form function editor) asks you to nominate a directory name. If it does not already exist you are prompted as to whether you want to create it:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_DIRECT) TYPE(*CHAR) LENGTH(70);
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_MBA) TYPE(*CHAR) LENGTH(1);
BEGIN_LOOP;
REQUEST FIELDS((#OV_DIRECT *NOID));
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(CHECK_DIR #OV_DIR
IF COND('#OV_RETC = OK');
MESSAGE MSGTXT('This directory already exists');
ELSE;

```

```

USE BUILTIN(OV_MESSAGE_BOX) WITH_ARGS('Do you want to create
IF COND(#OV_MBA = Y');
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(MAKE_DIR #OV_DIR)
IF COND(#OV_RETC *NE OK');
MESSAGE MSGTXT('Attempt to create directory failed');
ENDIF;
ENDIF;
ENDIF;
END_LOOP;

```

The following sample RDML function (which can be copied and pasted in the CS/400 free form function editor) asks you to nominate a file name and then indicates whether or not the file exists:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_DIRECT) TYPE(*CHAR) LENGTH(70);
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
BEGIN_LOOP;
MESSAGE MSGTXT('Specify name of file whose existence is to be checked');
REQUEST FIELDS((#OV_DIRECT *NOID));
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(CHECK_FILE #OV_DIRECT);
IF COND(#OV_RETC = OK');
MESSAGE MSGTXT('This file exists');
ELSE;
MESSAGE MSGTXT('This file does NOT exist');
ENDIF;
DISPLAY FIELDS((#OV_DIRECT *NOID));
END_LOOP;

```

The following sample RDML function (which can be copied and pasted in the CS/400 free form function editor) asks you to nominate a file name and whether the file should be set to read only status or normal (read/write) status:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_FILE) TYPE(*CHAR) LENGTH(70);
DEFINE FIELD(#OV_READ) TYPE(*CHAR) LENGTH(1) LABEL('Read O
DEFINE FIELD(#OV_NORM) TYPE(*CHAR) LENGTH(1) LABEL('Norma
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
*****;
BEGIN_LOOP;

```

```

MESSAGE MSGTXT('Specify name of file whose attribute is to be changed a
REQUEST FIELDS((#OV_FILE *L3 *P2 *NOID) (#OV_READ *L5 *P3) (#
IF COND('#ov_read = "1"');
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(SET_FILE #OV_FILE F
ELSE;
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(SET_FILE #OV_FILE N
ENDIF;
IF COND('#OV_RETC = OK');
MESSAGE MSGTXT('File attribute successfully changed');
ELSE;
MESSAGE MSGTXT('ERROR : File attribute was NOT changed');
ENDIF;
END_LOOP;

```

The following sample RDML function (which can be copied and pasted in the CS/400 free form function editor) asks you to nominate a from and to file name and then attempts to perform a copy operation:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_FROM) TYPE(*CHAR) LENGTH(60) DEFAULT('C:\
DEFINE FIELD(#OV_TO) TYPE(*CHAR) LENGTH(60) DEFAULT('C:\CO
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
*****.
BEGIN_LOOP;
MESSAGE MSGTXT('Specify the from and to file names');
REQUEST FIELDS(#OV_FROM #OV_TO);
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(COPY_FILE #OV_FRO
IF COND('#OV_RETC = OK');
MESSAGE MSGTXT('File copied');
ELSE;
MESSAGE MSGTXT('ERROR : File was NOT copied correctly');
ENDIF;
END_LOOP;

```

The following sample RDML function asks you to nominate a directory name and then retrieves and displays its contents. The resulting contents display can be sorted into various orders. This example can be copied and pasted into the CS/400 free form editor but the long REQUEST command may have to be "unfolded" before the function will be accepted as valid RDML code:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_DIRECT) TYPE(*CHAR) LENGTH(70);
DEFINE FIELD(#OV_FILTER) TYPE(*CHAR) LENGTH(3) LABEL('Option
DEFINE FIELD(#OV_BYTES) TYPE(*DEC) LENGTH(9) DECIMALS(0) L
DEFINE FIELD(#OV_OBJECT) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEF_COND NAME(*OBJECTS) COND('#ov_object *gt 0');
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_ERRN) TYPE(*DEC) LENGTH(15) DECIMALS(0);
DEFINE FIELD(#OV_NAME) TYPE(*CHAR) LENGTH(15);
DEFINE FIELD(#OV_PREFIX) TYPE(*CHAR) LENGTH(12);
DEFINE FIELD(#OV_SUFFIX) TYPE(*CHAR) LENGTH(3);
DEFINE FIELD(#OV_DATE) TYPE(*CHAR) LENGTH(8);
DEFINE FIELD(#OV_TIME) TYPE(*CHAR) LENGTH(6);
DEFINE FIELD(#OV_ISDIR) TYPE(*CHAR) LENGTH(1);
DEFINE FIELD(#OV_SIZE) TYPE(*DEC) LENGTH(9) DECIMALS(0) EDI
DEFINE FIELD(#OV_PB01) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEFINE FIELD(#OV_PB02) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEFINE FIELD(#OV_PB03) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEFINE FIELD(#OV_PB04) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEF_LIST NAME(#WLIST) FIELDS(#OV_NAME #OV_PREFIX #OV_SUF
DEF_LIST NAME(#DLIST) FIELDS(#OV_NAME #OV_PREFIX #OV_SUF
*****
EXECUTE SUBROUTINE(WTOD);
BEGIN_LOOP;
REQUEST FIELDS((#OV_DIRECT *L3 *P2 *NOID) (#OV_FILTER *L4 *P
      (#OV_PB04 *L8 *P38 *NOID *OBJECTS *IOCOND) (#OV_BYTE
CASE OF _FIELD(#IO$KEY);
WHEN VALUE_IS('= B1');
SORT_LIST NAMED(#WLIST) BY _FIELDS(#OV_NAME);
EXECUTE SUBROUTINE(WTOD);
WHEN VALUE_IS('= B2');
SORT_LIST NAMED(#WLIST) BY _FIELDS(#OV_SUFFIX #OV_PREFIX)
EXECUTE SUBROUTINE(WTOD);
WHEN VALUE_IS('= B3');
SORT_LIST NAMED(#WLIST) BY _FIELDS(#OV_DATE #OV_TIME);
EXECUTE SUBROUTINE(WTOD);
WHEN VALUE_IS('= B4');
SORT_LIST NAMED(#WLIST) BY _FIELDS(#OV_SIZE);
EXECUTE SUBROUTINE(WTOD);

```

```

OTHERWISE;
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(GET_DIR #OV_DIREC
IF COND('#OV_RETC = OK');
EXECUTE SUBROUTINE(WTOD);
ELSE;
MESSAGE MSGTXT('ERROR: Unable to list specified directory ');
ENDIF;
ENDCASE;
END_LOOP;
*****;
SUBROUTINE NAME(WTOD);
CLR_LIST NAMED(#DLIST);
CHANGE FIELD(#OV_BYTES) TO(0);
SELECTLIST NAMED(#WLIST);
CHANGE FIELD(#OV_BYTES) TO('#ov_bytes + #ov_size');
ADD_ENTRY TO_LIST(#DLIST) WITH_MODE(*DISPLAY);
ENDSELECT;
ENDROUTINE;

```

The following sample RDML function asks you to nominate a directory name and then retrieves and displays its contents. The resulting contents display can be sorted into various orders. By double clicking on a displayed file name you can delete it. File deletion requests must be confirmed by clicking "Yes" in a message box. This example can be copied and pasted into the CS/400 free form editor but the long REQUEST command may have to be "unfolded" before the function will be accepted as valid RDML code:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_DIRECT) TYPE(*CHAR) LENGTH(70);
DEFINE FIELD(#OV_PIRECT) REFFLD(#OV_DIRECT);
DEFINE FIELD(#OV_FILTER) TYPE(*CHAR) LENGTH(3) LABEL('Option
DEFINE FIELD(#OV_BYTES) TYPE(*DEC) LENGTH(9) DECIMALS(0) L
DEFINE FIELD(#OV_OBJECT) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OV_SELECT) TYPE(*DEC) LENGTH(7) DECIMALS(0);
DEF_COND NAME(*OBJECTS) COND('#ov_object *gt 0');
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_ERRN) TYPE(*DEC) LENGTH(15) DECIMALS(0);
DEFINE FIELD(#OV_NAME) TYPE(*CHAR) LENGTH(15);
DEFINE FIELD(#OV_PREFIX) TYPE(*CHAR) LENGTH(12);

```

```

DEFINE FIELD(#OV_SUFFIX) TYPE(*CHAR) LENGTH(3);
DEFINE FIELD(#OV_DATE) TYPE(*CHAR) LENGTH(8);
DEFINE FIELD(#OV_TIME) TYPE(*CHAR) LENGTH(6);
DEFINE FIELD(#OV_ISDIR) TYPE(*CHAR) LENGTH(1);
DEFINE FIELD(#OV_SIZE) TYPE(*DEC) LENGTH(9) DECIMALS(0) EDI
DEFINE FIELD(#OV_PB01) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEFINE FIELD(#OV_PB02) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEFINE FIELD(#OV_PB03) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEFINE FIELD(#OV_PB04) TYPE(*CHAR) LENGTH(30) INPUT_ATR(PE
DEF_LIST NAME(#WLIST) FIELDS(#OV_NAME #OV_PREFIX #OV_SUF
DEF_LIST NAME(#DLIST) FIELDS(#OV_NAME #OV_PREFIX #OV_SUF
*****.
BEGIN_LOOP;
CHANGE FIELD(#OV_PIRECT) TO(#OV_DIRECT);
REQUEST FIELDS((#OV_DIRECT *L3 *P2 *NOID) (#OV_FILTER *L4 *P
      (#OV_PB04 *L8 *P38 *NOID *OBJECTS *IOCOND) (#OV_BYTE
CASE OF_FIELD(#IO$KEY);
WHEN VALUE_IS('= B1');
SORT_LIST NAMED(#WLIST) BY_FIELDS(#OV_NAME);
EXECUTE SUBROUTINE(WTOD);
WHEN VALUE_IS('= B2');
SORT_LIST NAMED(#WLIST) BY_FIELDS(#OV_SUFFIX #OV_PREFIX)
EXECUTE SUBROUTINE(WTOD);
WHEN VALUE_IS('= B3');
SORT_LIST NAMED(#WLIST) BY_FIELDS(#OV_DATE #OV_TIME);
EXECUTE SUBROUTINE(WTOD);
WHEN VALUE_IS('= B4');
SORT_LIST NAMED(#WLIST) BY_FIELDS(#OV_SIZE);
EXECUTE SUBROUTINE(WTOD);
OTHERWISE;
IF COND('( #ov_pirect = #ov_direct ) *and ( #ov_select *gt 0 ) *and ( #ov_object
EXECUTE SUBROUTINE(DELETE_FIL);
ELSE;
EXECUTE SUBROUTINE(Load_dir);
ENDIF;
ENDCASE;
END_LOOP;
*****.
SUBROUTINE NAME(Load_dir);

```

```

USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(GET_DIR #OV_DIREC
IF COND(#OV_RETC = OK);
EXECUTE SUBROUTINE(WTOD);
ELSE;
MESSAGE MSGTXT('ERROR: Unable to list specified directory ');
ENDIF;
ENDROUTINE;
*****;
SUBROUTINE NAME(DELETE_FIL);
DEFINE FIELD(#OV_MBA) TYPE(*CHAR) LENGTH(1);
DEFINE FIELD(#OV_MSG) TYPE(*CHAR) LENGTH(100);
GET_ENTRY NUMBER(#OV_SELECT) FROM_LIST(#DLIST);
IF COND(#ov_isdir = Y);
MESSAGE MSGTXT('Selected object is a directory and cannot be deleted');
ELSE;
USE BUILTIN(BCONCAT) WITH_ARGS('Confirm that file' #OV_NAME 'is
USE BUILTIN(OV_MESSAGE_BOX) WITH_ARGS(#OV_MSG 'Delete File
IF COND(#OV_MBA = Y);
USE BUILTIN(TCONCAT) WITH_ARGS(#OV_DIRECT \| #OV_NAME) T
USE BUILTIN(OV_FILE_SERVICE) WITH_ARGS(REMOVE_FILE #OV_M
IF COND(#OV_RETC = OK);
EXECUTE SUBROUTINE(Load_DIR);
MESSAGE MSGTXT('File successfully deleted');
ELSE;
MESSAGE MSGTXT('ERROR : Attempt to delete file failed');
ENDIF;
ENDIF;
ENDIF;
ENDROUTINE;
*****;
SUBROUTINE NAME(WTOD);
CLR_LIST NAMED(#DLIST);
CHANGE FIELD(#OV_BYTES) TO(0);
SELECTLIST NAMED(#WLIST);
CHANGE FIELD(#OV_BYTES) TO('#ov_bytes + #ov_size');
ADD_ENTRY TO_LIST(#DLIST) WITH_MODE(*DISPLAY);
ENDSELECT;
ENDROUTINE;

```

OV_INDEXED_SPACE

⇒ **Note:** Built-In Function Rules

Allows you to define and manipulate an indexed space.

Warning: This function does not support RDMLX fields.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 989

DLL Required: U_BIF989.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Type of indexed space operation to be performed. Pass as one of: CREATE Create a new indexed space. INSERT Unconditionally insert a new entry into an indexed space. PUT Insert a new or update an existing entry in	1	50		

		<p>an indexed space.</p> <p>GET Get an entry from an indexed space.</p> <p>FIRST Get the first entry in an indexed space.</p> <p>NEXT Get the next entry in an indexed space.</p> <p>DESTROY Destroy an indexed space and free associated system resources.</p> <p>Note that this Built-In Function only validates and acts upon the first character of the requested index space operation (i.e. C,I,P,G,F,N,D) but to maximize RDML function readability it is recommended that you use the full words CREATE, INSERT, PUT, GET, FIRST, NEXT and DESTROY.</p>				
2	A	Req	<p>Definition string or indexed space, identifier (or handle). An identifier (or handle) is the value returned to you in argument 2 when you create a new indexed space that uniquely identifies the indexed space.</p> <p>When Arg 1 is Pass this argument as:</p> <p>CREATE The indexed space definition string. See the following information for details of definition strings.</p> <p>any other The identifier (or handle) of the indexed space that is to be used by the requested operation (the identifier or handle is the value returned in return value 2 when a new index space is created. It uniquely identifies the indexed space which you wish to use).</p>	1	256	

3	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
4	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
5	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
6	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
7	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
8	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
9	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
10	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		
11	A	Opt	Definition string continuation. Only valid for CREATE operations, ignored for other operations.	1	256		

Return Values

--	--	--	--	--	--	--	--

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Standard Return Code OK = Completed normally NR = No record found ER = Error occurred.	2	2		
2	A	Opt	Returned indexed space identifier or handle. This return value is mandatory when argument 1 is passed as CREATE because it returns the identifier (or handle) of the indexed space that was created.	10	10		

Technical Notes

Indexed Spaces and Indexed Space Definition Strings

When an index space is being created (i.e. CREATE is used in argument 1) then arguments 2 through 11 must specify a definition string for the indexed space.

Arguments 2 through 11 are concatenated (trailing blanks in each separate argument are ignored) to form a single definition string that must be formatted thus :

name keyword(value), name keyword(value), name keyword(value), nam

where:

- "name" is the name of a valid RDML field that is defined in, or referenced by, the RDML function that is creating the list.
- "keyword" specifies the use of the field specified in "name". It may be one of KEY, DATA, AVG, MAX, MIN, COUNT and SUM. If a keyword is not specified then DATA is assumed as a default keyword.
- "value" specifies the name of a valid RDML field that is defined in, or referenced by, the RDML function defining the list. It is used to specify, for certain keyword values only, the field upon which the keyword activity should take place. Thus "..., A SUM(B), .." defines that indexed space field A is to contain the SUM (or total) of field B. Likewise, the string "..., X AVG(Y), .." defines that indexed space field X is to contain the average of

field Y.

The "keywords" AVG, MAX, MIN and SUM require associated "values" and thus must be formatted as AVG(value), MAX(value), MIN(value) and SUM(value) where value is the name of a field defined in the invoking RDML function.

The "keywords" KEY, DATA and COUNT do not require associated "values" and they should only be specified as KEY(), DATA() and COUNT().

An indexed space is best visualized as a table or grid.

The definition string defines what the columns in the indexed space are to be and how they should be used. The following examples illustrate this concept:

Definition String: deptment key(), deptdes

Can be visualized as :

Department (DEPARTMENT)	Department Description (DEPTDESC)

where DEPARTMENT is the single key to each table or grid entry. Note that "deptdesc" adopts the default keyword "data()" in this example.

Definition String : deptment key(), section key(), secdesc

Can be visualized as

Department (DEPARTMENT)	Section (SECTION)	Section Description (SECDESC)

where DEPARTMENT is the single key to each table or grid entry.

Notes / Rules / Guidelines for use of OV_INDEXED_SPACE

- Indexed spaces have been primarily designed to support "batch" style functions that process large volumes of information. They have been designed to provide:
 - An optimized way of accumulating aggregate multi-level summary information.
 - An optimized way of randomly and repeatedly accessing large lists of information without the overhead of a database row access (i.e. you can load the required information into an indexed space at the start of your function and then repeatedly (re)access the indexed space more efficiently than you can by accessing the DBMS directly).

For examples of these types of usage please refer to the following examples.

- An indexed space must have at least one key field/column defined.
- An indexed space can have at most 20 key fields/columns defined.
- An indexed space must have at least one non-key field/column defined.
- An index space can have at most 100 non-key fields/columns defined.
- The aggregate byte length of all key fields/columns in an indexed space definition cannot exceed 16K. Note that if you are coming even remotely close to this limit then you should consult your product vendor about application design.
- The aggregate byte length of all non-key fields/columns in an indexed space definition entry cannot exceed 16K. Note that if you are coming even remotely close to this limit then you should consult your product vendor about application design.
- The significance of defined keys decreases with the order of their definition. Thus the definition string "aaa key(), bbb key(), ccc key(), xxx data(), yyy data()" defines "aaa" as the most significant key and "ccc" as the least significant key.
- Key definitions can take place at any point in the definition string, but it is customary to place them at the beginning.
- There is no effective limit to how many entries can be placed into an indexed space, but you must REMEMBER AT ALL TIMES that indexed spaces use allocated system memory. The more entries that exist in an index space the

more overhead you are placing on the consumption of a system resource. References in this section to "no effective limit" actually mean that you are effectively constrained by how much memory your system can viably allocate and use.

- The operations FIRST and NEXT support sequential key order access, however they can only be used with indexed spaces where there are less than 63KB / 6 (for Windows 3.1) or 32MB / 6 (other environments) entries in the indexed space. This is because the storage of the indexed entry's key always takes 6 bytes (regardless of the actual aggregate length of the key fields). If you try to use FIRST/NEXT sequential processing on an indexed space that is too large to support it you will receive a specific error message and your application will be aborted. GET, PUT and INSERT operations are not subjected to this limitation and can be used with no effective limit on the number of entries in the indexed space.
- An indexed space can only be used by the function that creates it. Although you can easily pass the identifier (or handle) of an indexed space to another function, any attempt by the other function to access the index space may lead to application failure and or unpredictable results. Do not attempt to do this. It will not work because when you CREATE an indexed space, a unique access plan to data fields stored in the creating function is formed. This access plan is unique to the creating function and cannot be effectively used by any other RDML function.
- You should use the DESTROY operation in your functions. However, all indexed spaces created by an RDML function are automatically destroyed when it terminates.
- If you are using multiple definition strings with a CREATE operation please remember that trailing blanks are ignored. Therefore:

```
USE OV_INDEXED_SPACE (CREATE 'A KEY(), B KEY(), C ' 'KEY(), D D
```

will cause a run time syntax error because the strings will be concatenated to form the definition string:

```
A KEY(), B KEY(), CKEY(), D DATA(), E DATA()
```

- Indexed space key fields are treated (and sorted) as pure binary data. The indexed space has no sense of the definition of a field as alpha, packed, signed, DBCS, etc and will not account for this in any operation.

- The PUT and INSERT operations are significantly different. A PUT operation checks whether an entry in the index exists with the specified key already. If it does, it is aggregated and updated as appropriate. If it does not exist, then a new entry is created. An INSERT operation does not check. It unconditionally creates a new entry.
This means that if you wish to fill an indexed space from a DBMS table when you know you are creating unique entries, then using INSERT is substantially more efficient than using PUT.
- You can put duplicate keys into an indexed space (i.e., 2 or more entries that have exactly the same key values) by using the INSERT operation. However, the effects of doing this and/or the order in which the duplicates are processed is unspecified and may vary from platform to platform.
- The aggregation operations AVG(), MAX(), MIN(), SUM() and COUNT() are all performed with a maximum of 15 significant digits of precision.

You must not use arrays or array indices in indexed spaces.

Examples

The following sample RDML function (which can be copy and pasted into the L4W free form function editor) is designed to illustrate the relative efficiency of indexed spaces for random access. You can use it to create a simple indexed space of up to 100,000 entries, and then cause it to lookup each entry individually:

```

FUNCTION OPTIONS(*LIGHTUSAGE *DIRECT);
***** COMMENT(Define the index space columns);
DEFINE FIELD(#OV_KEY01) TYPE(*DEC) LENGTH(7) DECIMALS(0) E
DEFINE FIELD(#OV_KEY02) TYPE(*DEC) LENGTH(7) DECIMALS(0) E
DEFINE FIELD(#OV_KEY03) TYPE(*DEC) LENGTH(7) DECIMALS(0) E
DEFINE FIELD(#OV_DATA01) TYPE(*DEC) LENGTH(7) DECIMALS(0);
DEFINE FIELD(#OV_DATA02) TYPE(*DEC) LENGTH(7) DECIMALS(0);
DEFINE FIELD(#OV_DATA03) TYPE(*DEC) LENGTH(7) DECIMALS(0);
DEFINE FIELD(#OV_TOTAL) TYPE(*DEC) LENGTH(7) DECIMALS(0) E
***** COMMENT(Define the loop test limits);
DEFINE FIELD(#OV_MKEY01) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OV_MKEY02) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OV_MKEY03) TYPE(*DEC) LENGTH(7) DECIMALS(0)
***** COMMENT(Define other variables);
DEFINE FIELD(#OV_RC) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_HANDLE) TYPE(*CHAR) LENGTH(10);

```

```

***** COMMENT(Create the indexed space);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE 'ov_key01 l
***** COMMENT(Request details of how the list is to be initialized);
REQUEST FIELDS(#OV_MKEY01 #OV_MKEY02 #OV_MKEY03);
BEGINCHECK;
CONDCHECK FIELD(#OV_MKEY01) COND('((#OV_MKEY01 * #OV_MI
ENDCHECK;
***** COMMENT(Initialize the indexed space );
CHANGE FIELD(#OV_TOTAL) TO(0);
BEGIN_LOOP USING(#OV_KEY01) TO(#OV_MKEY01);
CHANGE FIELD(#OV_DATA01) TO(#OV_KEY01);
BEGIN_LOOP USING(#OV_KEY02) TO(#OV_MKEY02);
CHANGE FIELD(#OV_DATA02) TO(#OV_KEY02);
BEGIN_LOOP USING(#OV_KEY03) TO(#OV_MKEY03);
CHANGE FIELD(#OV_DATA03) TO(#OV_KEY03);
CHANGE FIELD(#OV_TOTAL) TO('#OV_TOTAL + 1');
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(INSERT #OV_HANI
END_LOOP;
END_LOOP;
END_LOOP;
*****;
MESSAGE MSGTXT('Index area initialized. Total entry's is shown. Click OK
DISPLAY FIELDS(#OV_TOTAL);
*****;
CHANGE FIELD(#OV_TOTAL) TO(0);
BEGIN_LOOP USING(#OV_KEY01) TO(#OV_MKEY01);
BEGIN_LOOP USING(#OV_KEY02) TO(#OV_MKEY02);
BEGIN_LOOP USING(#OV_KEY03) TO(#OV_MKEY03);
CHANGE FIELD(#OV_TOTAL) TO('#OV_TOTAL + 1');
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(GET #OV_HANDLE
IF COND('#OV_RC *ne OK) *or (#OV_DATA01 *ne #OV_key01) *or (#O'
MESSAGE MSGTXT('Lookup was error was detected for key values shown');
REQUEST FIELDS(#OV_KEY01 #OV_KEY02 #OV_KEY03 #OV_RC);
ENDIF;
END_LOOP;
END_LOOP;
END_LOOP;
MESSAGE MSGTXT('Test completed. Total number of lookup tests is shown'
DISPLAY FIELDS(#OV_TOTAL);

```

The following sample RDML function uses an indexed space to aggregate details of employee salary information. It uses the standard LANSA demonstration file PSLMST as the basis of employee salary information:

```

FUNCTION OPTIONS(*LIGHTUSAGE *DIRECT);
***** COMMENT(Departmental Summary definitions);
DEFINE FIELD(#OVTDEPSAL) TYPE(*DEC) LENGTH(15) DECIMALS(2)
DEFINE FIELD(#OVXDEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVNDEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVADEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVCDEPSAL) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OVFDEPSAL) TYPE(*CHAR) LENGTH(256) DECIMALS(0)
CHANGE FIELD(#OVFDEPSAL) TO('deptment key(), ovtdepsal sum(salary)');
DEFINE FIELD(#OVHDEPSAL) TYPE(*CHAR) LENGTH(10) LABEL('Sp');
DEF_LIST NAME(#OVSDEPSAL) FIELDS(#DEPARTMENT #OVTDEPSAL #OVXDEPSAL #OVNDEPSAL #OVADEPSAL #OVCDEPSAL #OVFDEPSAL #OVHDEPSAL);
*****;
DEFINE FIELD(#OV_RC) TYPE(*CHAR) LENGTH(2) LABEL('Return Code');
***** COMMENT(Create the indexed space);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OVFDEPSAL);
***** COMMENT((Pass over the data and update the summary details)');
SELECT FIELDS(#DEPARTMENT #SALARY) FROM_FILE(PSLMST);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(PUT #OVHDEPSAL);
ENDSELECT;
***** COMMENT(Now load/show a browse list with the results);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(FIRST #OVHDEPSAL);
DOWHILE COND('#OV_RC = OK');
ADD_ENTRY TO_LIST(#OVSDEPSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(NEXT #OVHDEPSAL);
ENDWHILE;
DISPLAY BROWSELIST(#OVSDEPSAL);
***** COMMENT(Destroy the indexed space);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(DESTROY #OVHDEPSAL);

```

The following sample RDML function is identical to the previous one except that it uses a second indexed space to aggregate information for all departments and then adds the "grand" aggregates to the end of the aggregate list that is displayed to the user. This demonstrates how indexed spaces can be used to perform multiple level totaling by using multiple indexed spaces :

```

FUNCTION OPTIONS(*LIGHTUSAGE *DIRECT);
***** COMMENT(Departmental Summary definitions);
DEFINE FIELD(#OVTDEPSAL) TYPE(*DEC) LENGTH(15) DECIMALS(2)
DEFINE FIELD(#OVXDEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVNDEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVADEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVCDEPSAL) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OVFDEPSAL) TYPE(*CHAR) LENGTH(256) DECIMALS(0)
CHANGE FIELD(#OVFDEPSAL) TO('deptment key(), ovtdepsal sum(salary)');
DEFINE FIELD(#OVHDEPSAL) TYPE(*CHAR) LENGTH(10) LABEL('Space');
DEFINE FIELD(#OVSDEPSAL) TYPE(*CHAR) LENGTH(10) LABEL('Department Name');
*****
DEFINE FIELD(#OVFGRAND) TYPE(*CHAR) LENGTH(256) DECIMALS(0)
CHANGE FIELD(#OVFGRAND) TO('ovkgrand key(), ovtdepsal sum(salary)');
DEFINE FIELD(#OVHGRAND) TYPE(*CHAR) LENGTH(10) LABEL('Space');
DEFINE FIELD(#OVKGRAND) TYPE(*CHAR) LENGTH(10) LABEL('Invariant');
*****
DEFINE FIELD(#OV_RC) TYPE(*CHAR) LENGTH(2) LABEL('Return Code');
***** COMMENT(Create the indexed spaces);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OVFDEPSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OVFGRAND);
***** COMMENT((Pass over the data and create the summary indexes)');
SELECT FIELDS(#DEPARTMENT #SALARY) FROM_FILE(PSLMST);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(PUT #OVHDEPSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(PUT #OVHGRAND);
ENDSELECT;
***** COMMENT(Now load/show a browse list with the results);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(FIRST #OVHDEPSAL);
DOWHILE COND('#OV_RC = OK');
ADD_ENTRY TO_LIST(#OVSDEPSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(NEXT #OVHDEPSAL);
ENDWHILE;
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(GET #OVHGRAND);
CHANGE FIELD(#DEPARTMENT) TO(#OVKGRAND);
ADD_ENTRY TO_LIST(#OVSDEPSAL);
***** COMMENT(Display the results);
DISPLAY BROWSELIST(#OVSDEPSAL);
***** COMMENT(Destroy the indexed space);

```

```
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(DESTROY #OVHDF
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(DESTROY #OVHGI
```

The following example RDML function is again very similar to the previous two, except that it produces two additional aggregation lists by using two additional indexed spaces. The second is by department/section and the third is by salary (i.e., a distribution of how many employees earn a particular salary value) :

```
FUNCTION OPTIONS(*LIGHTUSAGE *DIRECT);
***** COMMENT(Departmental Summary definitions);
DEFINE FIELD(#OVTDEPSAL) TYPE(*DEC) LENGTH(15) DECIMALS(2)
DEFINE FIELD(#OVXDEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVNDEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVADEPSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVCDEPSAL) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OVFDEPSAL) TYPE(*CHAR) LENGTH(256) DECIMALS(0)
CHANGE FIELD(#OVFDEPSAL) TO('deptment key(), ovtdepsal sum(salary)');
DEFINE FIELD(#OVHDEPSAL) TYPE(*CHAR) LENGTH(10) LABEL('Sp:');
DEF_LIST NAME(#OVSDEPSAL) FIELDS(#DEPARTMENT #OVTDEPSAL #OVXDEPSAL #OVNDEPSAL #OVADEPSAL #OVCDEPSAL #OVFDEPSAL #OVHDEPSAL);
***** COMMENT(Section Summary definitions);
DEFINE FIELD(#OVTSECSAL) TYPE(*DEC) LENGTH(15) DECIMALS(2)
DEFINE FIELD(#OVXSECSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVNSECSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVASECSAL) TYPE(*DEC) LENGTH(11) DECIMALS(2)
DEFINE FIELD(#OVCSECSAL) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OVFSECSAL) TYPE(*CHAR) LENGTH(256) DECIMALS(0)
CHANGE FIELD(#OVFSECSAL) TO('deptment key(), section key(), ovtdepsal sum(salary)');
DEFINE FIELD(#OVHSECSAL) TYPE(*CHAR) LENGTH(10) LABEL('Sp:');
DEF_LIST NAME(#OVSSECSAL) FIELDS(#DEPARTMENT #SECTION #OVTSECSAL #OVXSECSAL #OVNSECSAL #OVASECSAL #OVCSECSAL #OVFSECSAL #OVHSECSAL);
***** COMMENT(Salary Distribution Definitions);
DEFINE FIELD(#OVCSALSAL) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE FIELD(#OVFSALSAL) TYPE(*CHAR) LENGTH(256) DECIMALS(0)
CHANGE FIELD(#OVFSALSAL) TO('salary key(), ovcsalsal count()');
DEFINE FIELD(#OVHSALSAL) TYPE(*CHAR) LENGTH(10) LABEL('Sp:');
DEF_LIST NAME(#OVSSALSAL) FIELDS(#SALARY #OVCSALSAL #OVFSALSAL #OVHSALSAL);
*****;
DEFINE FIELD(#OV_RC) TYPE(*CHAR) LENGTH(2) LABEL('Return Code');
***** COMMENT(Create the indexed space);
```

```

USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OVFDEPSA);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OVFSECSA);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OVFSALSAL);
***** COMMENT((Pass over the data and create the summary indexes');
SELECT FIELDS(#DEPARTMENT #SECTION #SALARY) FROM_FILE(PSLM);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(PUT #OVHDEPSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(PUT #OVHSECSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(PUT #OVHSALSAL);
ENDSELECT;
***** COMMENT(Now load/show a browse list with the results);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(FIRST #OVHDEPSA);
DOWHILE COND('#OV_RC = OK');
ADD_ENTRY TO_LIST(#OVSDEPSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(NEXT #OVHDEPSA);
ENDWHILE;
DISPLAY BROWSELIST(#OVSDEPSAL);
***** COMMENT(Now load/show a browse list with the results);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(FIRST #OVHSECSA);
DOWHILE COND('#OV_RC = OK');
ADD_ENTRY TO_LIST(#OVSSSECSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(NEXT #OVHSECSA);
ENDWHILE;
DISPLAY BROWSELIST(#OVSSSECSAL);
***** COMMENT(Now load/show a browse list with the results);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(FIRST #OVHSALSAL);
DOWHILE COND('#OV_RC = OK');
ADD_ENTRY TO_LIST(#OVSSALSAL);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(NEXT #OVHSALSAL);
ENDWHILE;
DISPLAY BROWSELIST(#OVSSALSAL);
***** COMMENT(Destroy the indexed spaces);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(DESTROY #OVHDEPSA);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(DESTROY #OVHSECSA);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(DESTROY #OVHSALSAL);

```

The following sample RDML function is designed to demonstrate how indexed lists can be used to improve application performance in "batch" style jobs processing large amounts of information.

Logically, the "engine" loop of this function is like this :

```
BEGIN_LOOP TO(#OV_ITER);
  SELECT FIELDS(#EMPNO #SURNAME #GIVENAME #DEPARTMENT #
    FETCH FIELDS(#DEPTDESC) FROM_FILE(DEPTAB) WITH_KEY(#
    FETCH FIELDS(#SECDESC) FROM_FILE(SECTAB) WITH_KEY(#D
  ENDSELECT;
END_LOOP;
```

This loop selects all the employees in the standard shipped demonstration table PSLMST (repeated for a specified number of iterations). It does this to attempt to emulate the processing of a large number of records typically found in a "batch" job.

For each row selected it fetches in the associated department and section description from the DEPTAB and SECTAB tables.

However, the "engine" loop has actually been coded as:

```
BEGIN_LOOP TO(#OV_ITER);
  SELECT FIELDS(#EMPNO #SURNAME #GIVENAME #DEPARTMENT #
    IF COND(*USEINDEX);
    USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(GET #OV_DI
    USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(GET #OV_SE
  ELSE;
    FETCH FIELDS(#DEPTDESC) FROM_FILE(DEPTAB) WITH_KEY
    FETCH FIELDS(#SECDESC) FROM_FILE(SECTAB) WITH_KEY(#
  ENDIF;
  ENDSELECT;
END_LOOP;
```

which allows you to run the engine loop using either the DBMS (i.e., the FETCH commands) or an indexed space (i.e., the USE OV_INDEXED_SPACE commands) to get the department and section descriptions. By doing this you will be able to see the speed advantage that an indexed space provides over performing a full DBMS access to find information.

The full sample RDML function follows.

Notice that you can run the "engine" loop through 1 -> 20 iterations using either the DBMS (specify D) or an indexed space (specify I) to fetch the department and section description details. All department and section descriptions are loaded into their associated indexed spaces at the beginning of the function:

```

FUNCTION OPTIONS(*LIGHTUSAGE *DIRECT);
***** COMMENT(Define and load the department indexed space);
DEFINE FIELD(#OV_DEPDEF) TYPE(*CHAR) LENGTH(50) LABEL('Dept');
CHANGE FIELD(#OV_DEPDEF) TO('deptment key(), deptdesc');
DEFINE FIELD(#OV_DEPTAB) TYPE(*CHAR) LENGTH(10) LABEL('Dept');
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OV_DEPTAB);
SELECT FIELDS(#DEPARTMENT #DEPTDESC) FROM_FILE(DEPTAB);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(INSERT #OV_DEPTAB);
ENDSELECT;
***** COMMENT(Define and load the section indexed space);
DEFINE FIELD(#OV_SECDEF) TYPE(*CHAR) LENGTH(50) LABEL('Section');
CHANGE FIELD(#OV_SECDEF) TO('deptment key(), section key(), secdesc');
DEFINE FIELD(#OV_SECTAB) TYPE(*CHAR) LENGTH(10) LABEL('Section');
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(CREATE #OV_SECTAB);
SELECT FIELDS(#DEPARTMENT #SECTION #SECDESC) FROM_FILE(SECTAB);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(INSERT #OV_SECTAB);
ENDSELECT;
***** COMMENT(Define other variables);
OVERRIDE FIELD(#GIVENAME) LENGTH(10);
DEFINE FIELD(#OV_RC) TYPE(*CHAR) LENGTH(2) LABEL('Return Code');
DEFINE FIELD(#OV_MODE) TYPE(*CHAR) LENGTH(1) LABEL('Mode');
DEFINE FIELD(#OV_ITER) TYPE(*DEC) LENGTH(3) DECIMALS(0) LABEL('Iterations');
DEFINE FIELD(#OV_TOTAL) TYPE(*DEC) LENGTH(7) DECIMALS(0) LABEL('Total');
DEF_COND NAME(*USEINDEX) COND('#OV_MODE = I');
DEF_LIST NAME(#OV_LIST) FIELDS(#SURNAME #GIVENAME #DEPARTMENT);
***** COMMENT(Repeat testing until cancelled);
BEGIN_LOOP;
***** COMMENT(Request and validate testing details);
POP_UP FIELDS((#OV_MODE *IN) (#OV_ITER *IN)) EXIT_KEY(*NO) PRT;
BEGINCHECK;
VALUECHECK FIELD(#OV_MODE) WITH_LIST(D I) MSGTXT('Mode must be D or I');
RANGECHECK FIELD(#OV_ITER) RANGE((1 20)) MSGTXT(('Number of iterations must be between 1 and 20'));
ENDCHECK;
***** COMMENT(Repeat the test for the number of iterations);
CLR_LIST NAMED(#OV_LIST);
BEGIN_LOOP TO(#OV_ITER);
SELECT FIELDS(#EMPNO #SURNAME #GIVENAME #DEPARTMENT #SECTION);
IF COND(*USEINDEX);
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(GET #OV_DEPTAB);

```

```
USE BUILTIN(OV_INDEXED_SPACE) WITH_ARGS(GET #OV_SECTAB)
ELSE;
FETCH FIELDS(#DEPTDESC) FROM_FILE(DEPTAB) WITH_KEY(#DEPT
FETCH FIELDS(#SECDESC) FROM_FILE(SECTAB) WITH_KEY(#DEPTM
ENDIF;
ADD_ENTRY TO_LIST(#OV_LIST);
ENDSELECT;
END_LOOP;
***** COMMENT(display the results);
DISPLAY FIELDS(#OV_TOTAL) BROWSELIST(#OV_LIST) EXIT_KEY(
END_LOOP;
```

OV_MESSAGE_BOX

⇒ **Note:** Built-In Function Rules

Present a message box.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 998

DLL Required: U_BIF998.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Text to display in message.	1	256		
2	A	Opt	Title to display in message box. Default is the function description	1	40		
3	A	Opt	Button(s) to be enabled. O = OK OC = OK & CANCEL C = OK (CANCEL in OS2 operating system, which is no longer supported) E = OK (ENTER in OS2 operating system, which is no longer supported)	1	3		

			EC = OK & CANCEL (ENTER & CANCEL in OS2 operating system, which is no longer supported) RC = RETRY & CANCEL ARI = ABORT, RETRY & IGNORE YN = YES & NO Default is "O"				
4	A	Opt	Icon to be shown in message box. B1 = button 1 B2 = button 2 B3 = button 3 Default is "B1"	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Button that was used by the user E = ENTER O = OKAY C = CANCEL A = ABORT R = RETRY I = IGNORE Y = YES N = NO X = error	1	1		

OV_PASTE_CLIPBOARD

⇒ **Note:** Built-In Function Rules

Paste from the clipboard to a working list.

Warning:: This function does not support RDMLX fields.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 996

DLL Required: U_BIF996.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	The field delimiter that was used. T = tab character C = comma character Other = blank character Default is 'T'.	1	1		
2	A	Opt	Alphanumeric fields were quoted. Y = yes	1	1		

		N = no Default is 'Y'				
--	--	--------------------------	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	List	Req	The name of the working list whose entries are to be pasted from the clipboard.	2	10		
2	A	Req	Return value indicating if the paste to the list was successful. OK = paste was successful ER = paste was unsuccessful	2	2		

Technical Notes

New line characters in the data pasted from the clipboard represent the end of each working list entry.

List entry fields that are not present in the data will be initialized to blanks or zeroes.

OV_POST_CLIPBOARD

⇒ **Note:** Built-In Function Rules

Post a working list to the clipboard.

Warning: This function does not support RDMLX fields.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 995

DLL Required: U_BIF995.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	List	Req	The name of the working list whose entries are to be posted to the clipboard.				
2	A	Opt	The field delimiter to be used: T = tab character C = comma character N = new line character Other = blank character Default is 'T'.	1	1		
3	A	Opt	Alphanumeric fields are to be quoted.	1	1		

			Y = yes N = no Default is 'Y'				
4	A	Opt	Append Line Feed Y = yes. Append Line Feed after each entry. N = no. Do not append Line Feed to the last entry. Default is 'Y'.	1	1		

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Return value indicating if the post of the list was successful. OK = post was successful ER = post was unsuccessful.	2	2		

OV_QUERY_SYS_INFO

⇒ **Note:** Built-In Function Rules

Query system configuration information.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 993

DLL Required: U_BIF993.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Information to be queried Pass as one of : SYS_DRIV SYS_DIR SYS_DIR_EXECUTE SYS_DIR_SOURCE SYS_DIR_OBJECT PART_DRIV PART_DIR	1	256		

			PART_DIR_EXECUTE PART_DIR_SOURCE PART_DIR_OBJECT TEMP_DRIV TEMP_DIR DRIV_LIST				
--	--	--	--	--	--	--	--

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Ma Dec
1	A	Req	Information returned. SYS_DRIV Returns the drive in which the LANSAsystem is located as a drive letter followed by a colon. e.g: C: or D: or E: SYS_DIR Returns the root directory in which the LANSAsystem is located. e.g: D:\X_WIN95\X_LANSAs\ SYS_DIR_EXECUTE Returns the directory in which the LANSAsystem EXECUTE objects are located. e.g: D:\X_WIN95\X_LANSAs\EXECUTE\ SYS_DIR_SOURCE Returns the directory in which the LANSAsystem SOURCE objects are located. e.g: D:\X_WIN95\X_LANSAs\SOURCE\ SYS_DIR_OBJECT Returns the directory in which the LANSAsystem OBJECT objects are located. e.g: D:\X_WIN95\X_LANSAs\OBJECT\	1	256		

			<p>PART_DRIV</p> <p>Returns the drive in which the current partitions LANSA system is located as a drive letter followed by a colon. e.g: C: or D: or E:</p> <p>PART_DIR</p> <p>Returns the root directory in which the current partitions system is located. e.g.: D:\X_WIN95\X_LANSA\X_DEM\</p> <p>PART_DIR_EXECUTE</p> <p>Returns the directory in which the current partitions EXECUTE objects are located. e.g: D:\X_WIN95\X_LANSA\X_DEM\EXECUTE\</p> <p>PART_DIR_SOURCE</p> <p>Returns the directory in which the current partitions SOURCE objects are located. e.g: D:\X_WIN95\X_LANSA\X_DEM\SOURCE\</p> <p>PART_DIR_OBJECT</p> <p>Returns the directory in which the current partitions OBJECT objects are located. e.g: D:\X_WIN95\X_LANSA\X_DEM\OBJECT\</p> <p>TEMP_DRIV</p> <p>Returns the drive in which temporary files / objects should be created as a drive letter followed by a colon. e.g: C: or D: or E:</p> <p>TEMP_DIR</p> <p>Returns the directory in which temporary files/objects should be created. e.g: D:\TEMP\</p> <p>DRIV_LIST</p> <p>Returns a working list in return value 2. This value (return value 1) must be passed as a dummy argument when making a DRIV_LIST request.</p>			
2	List	Opt	Working list to contained returned information. Currently only required for a DRIV_LIST request. Refer to the following examples for the	N/A		

		layout and format of the returned working list for DRIV_LIST requests.				
--	--	---	--	--	--	--

Examples

The following sample RDML function (which can be copied and pasted in the CS/400 free form function editor) queries and displays all possible drive/path combinations:

```
FUNCTION OPTIONS(*DIRECT);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(SYS_DRIV);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(SYS_DIR);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(SYS_DIR_EXEC);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(SYS_DIR_OBJ);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(SYS_DIR_SOU);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(PART_DRIV);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(PART_DIR);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(PART_DIR_EXEC);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(PART_DIR_OBJ);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(PART_DIR_SOU);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(TEMP_DRIV);
EXECUTE SUBROUTINE(SHOW_INFO) WITH_PARMS(TEMP_DIR);
***** COMMENT(Display subroutine);
SUBROUTINE NAME(SHOW_INFO) PARMS( OV_QUERY);
DEFINE FIELD(#OV_QUERY) TYPE(*CHAR) LENGTH(50);
DEFINE FIELD(#OV_RESULT) TYPE(*CHAR) LENGTH(50);
USE BUILTIN(OV_QUERY_SYS_INFO) WITH_ARGS(#OV_QUERY) TO_
DISPLAY_FIELDS(#OV_QUERY #OV_RESULT);
ENDROUTINE;
```

This sample displays the drive letters and drive types of all disk drives attached to the current PC. Note that the drive type is returned as REM (Removable drive), FIX (Fixed drive), NET (Network drive), CD (CD-ROM drive) or RAM (RAM Drive).

Under Windows 3.1 the available drives are only classified as type REM, FIX or NET.

Under Windows 95/NT the available drives are only classified as type REM,

FIX, NET, CD or RAM.

Note also that in this sample the drive letter is returned as a char(2) in format A:, B:, etc :

```
FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_RESULT) TYPE(*CHAR) LENGTH(50);
DEFINE FIELD(#OV_DRIVE) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_TYPE) TYPE(*CHAR) LENGTH(3);
DEF_LIST NAME(#OV_DRIVD) FIELDS(#OV_DRIVE #OV_TYPE);
DEF_LIST NAME(#OV_DRIVW) FIELDS(#OV_DRIVE #OV_TYPE) TYPE(*CHAR) LENGTH(2);
***** COMMENT(Extract list off drives and display);
USE BUILTIN(OV_QUERY_SYS_INFO) WITH_ARGS(DRIV_LIST) TO_C
CLR_LIST NAMED(#OV_DRIVD);
SELECTLIST NAMED(#OV_DRIVW);
ADD_ENTRY TO_LIST(#OV_DRIVD);
ENDSELECT;
DISPLAY BROWSELIST(#OV_DRIVD);
```

This sample requests that you specify the drive type you are interested in as REM, FIX, NET or ALL and displays all drives of the requested type in a drop down. Note that in this sample the drive letter is returned as a char(1) in format A, B, C, etc :

```
FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_RESULT) TYPE(*CHAR) LENGTH(50);
DEFINE FIELD(#OV_DRIVE) TYPE(*CHAR) LENGTH(1) INPUT_ATR(D);
DEFINE FIELD(#OV_WDRIVE) TYPE(*CHAR) LENGTH(1);
DEFINE FIELD(#OV_TYPE) TYPE(*CHAR) LENGTH(3);
DEFINE FIELD(#OV_WTYPE) TYPE(*CHAR) LENGTH(3);
DEFINE FIELD(#OV_RC) TYPE(*CHAR) LENGTH(2);
DEFINE FIELD(#OV_WCOUNT) TYPE(*DEC) LENGTH(7) DECIMALS(0);
DEF_LIST NAME(#OV_DRIVW) FIELDS(#OV_WDRIVE #OV_WTYPE) TYPE(*CHAR) LENGTH(1);
DEF_COND NAME(*SHOWDRIV) COND('#OV_wcount *gt 0');
***** COMMENT(Request type to be shown in Drop Down);
BEGIN_LOOP;
REQUEST FIELDS(#OV_TYPE (#OV_DRIVE *SHOWDRIV));
BEGINCHECK;
VALUECHECK FIELD(#OV_TYPE) WITH_LIST('REM' 'FIX' 'NET' 'ALL');
ENDCHECK;
***** COMMENT(Extract list off drives and display);
```

```
USE BUILTIN(OV_QUERY_SYS_INFO) WITH_ARGS(DRIV_LIST) TO_C
USE BUILTIN(DROP_DD_VALUES) WITH_ARGS(DDHD) TO_GET(#OV
CHANGE FIELD(#OV_WCOUNT #OV_DRIVE) TO(*NULL);
SELECTLIST NAMED(#OV_DRIVW);
IF COND('#OV_type = #OV_wtype) *or (#OV_type = ALL)');
CHANGE FIELD(#OV_WCOUNT) TO('#OV_wcount + 1');
USE BUILTIN(ADD_DD_VALUES) WITH_ARGS(DDHD *BLANKS #OV_
IF_NULL FIELD(#OV_DRIVE);
CHANGE FIELD(#OV_DRIVE) TO(#OV_WDRIVE);
ENDIF;
ENDIF;
ENDSELECT;
IF COND('#OV_wcount <= 0');
MESSAGE MSGTXT('No drives of the requested type exist on (or are accessi
ENDIF;
END_LOOP;
```

OV_SLEEP

⇒ **Note:** Built-In Function Rules

Pauses the program for the specified number of milliseconds.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 986

DLL Required: U_BIF986.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	YES
LANSA for i	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	N	Req	Time to wait in milliseconds	1	15	0	0

Return Values

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Empty is always returned.	2	2		

Technical Notes

On Linux the time to wait is converted from milliseconds to seconds.

Example

To sleep in 5 seconds, use this command:

```
USE BUILTIN(OV_SLEEP) WITH_ARGS(5000)
```

OV_SOUND_ALARM

⇒ **Note:** Built-In Function Rules

Sound the device alarm.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 999

DLL Required: U_BIF999.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/ Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Alarm option. W = warning E = error N = note Default is 'N'	1	1		

Return Values

No Return Values

OV_SYSTEM_SERVICE

⇒ **Note:** Built-In Function Rules

Performs a basic system service.

Note: The user of this Built-In Function is responsible for any impact it has on any application. No warranty of any kind is expressed or implied. Refer to full [Disclaimer](#).

Function No: 991

DLL Required: U_BIF991.DLL

For use with

Visual LANSA for Windows	YES
Visual LANSA for Linux	NO
LANSA for i	NO

Arguments

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Req	Type of system service required Pass as one of: START Execute another (non-LANSA) application. START_LANSA Start another LANSA application executing. START_UNIQUE Execute another (non-LANSA) application if not already started.	1	50		
2	A	Opt	Requested Service Argument 1	1	256		

When Arg 1 is

Pass this argument as:

START

Name of application (.EXE) to be started.

You may use a fully qualified name (i.e. with path information) or just the .EXE name. If you do not suffix the name with .EXE then the .EXE suffix will be automatically added.

START_LANSA

Name of the LANSA process, process and function or form that is to be started as another independent system process / job.

Pass this value as either

PROC=PPPPPPPPPP, or,

PROC=PPPPPPPPPP FUNC=FFFFFFF, or,

FORM=OOOOOOOOOO

only (where PPPPPPPPP is the process name, FFFFFFFF is the function name and OOOOOOOOO is the form name).

Do NOT under any circumstances include any other

X_RUN parameters into this argument (e.g: PART=PPP, DBID=XXXX, etc).

START_UNIQUE

Name of application (.EXE) to be started if it is not already started. This service is only available in MS Windows environments.

You should NOT use a fully qualified name (i.e. with path information), just the .EXE name. If you do not suffix the name with .EXE then the .EXE suffix will be automatically added.

Note:

If Arg 1 is START or START_UNIQUE and arg 2 is LCOADM32 or LCOADM32.EXE

			(ie without the fully qualified path), LCOADM32.EXE will be found in the LANSA\Connect directory.				
3	A	Opt	Requested Service Argument 2 When Arg 1 is Pass this argument as: START Parameters to pass to application being started. LANSA This argument is not required and any value specified will be ignored. START_UNIQUE Parameters to pass to the application if it needs to be started.	1	256		

Return Values

No	Type	Req/Opt	Description	Min Len	Max Len	Min Dec	Max Dec
1	A	Opt	Basic Return Code OK = Completed (normally) ER = Error occurred.	2	2		
2	N	Opt	Extended Error Code This is the operating system error code (when available) that may aid you in error handling or error reporting.				
3	List	Opt	Returned as OK = Completed normally ER = Error occurred. Only returned for certain argument 1 values as follows: START				

		Not returned.				
		START_LANSA				
		Not returned.				
		START_UNIQUE				
		Not returned.				

Example

The following sample RDML function (which can be copied and pasted into the L4W free form function editor) requests that you specify a program (.EXE) name and any parameters to be passed to it. An attempt is then made to start the specified program executing:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_PGM) TYPE(*CHAR) LENGTH(50) LABEL('Program');
DEFINE FIELD(#OV_PARMS) TYPE(*CHAR) LENGTH(50) LABEL('Parameters');
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2) LABEL('Return Code');
DEFINE FIELD(#OV_ERRN) TYPE(*DEC) LENGTH(7) DECIMALS(0) LABEL('Error Number');
*****
BEGIN_LOOP;
REQUEST FIELDS(#OV_PGM #OV_PARMS (#OV_RETC *OUT) (#OV_ERRN *OUT));
USE BUILTIN(OV_SYSTEM_SERVICE) WITH_ARGS(START #OV_PGM #OV_PARMS);
IF COND(#OV_RETC = OK);
MESSAGE MSGTXT('Program started');
ELSE;
MESSAGE MSGTXT('Error detected when attempting to start program');
ENDIF;
END_LOOP;

```

The following sample RDML function requests that you specify up to 100 lines of text. The lines you specify are transferred into a file, and then either the EPM (OS/2) or NOTEPAD (Windows) source line editors are started against the file created:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_TEMP) TYPE(*CHAR) LENGTH(20) LABEL('Temp File Name');
DEFINE FIELD(#OV_EDITOR) TYPE(*CHAR) LENGTH(20) LABEL('Editor Name');

```

```

IF COND(*CPUTYPE = OS2);
CHANGE FIELD(#OV_EDITOR) TO(EPM);
ELSE;
CHANGE FIELD(#OV_EDITOR) TO(NOTEPAD);
ENDIF;
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2) LABEL('Return (
DEFINE FIELD(#OV_TEXT) TYPE(*CHAR) LENGTH(70) COLHDG('Text
DEF_LIST NAME(#OV_LISTD) FIELDS(#OV_TEXT);
DEF_LIST NAME(#OV_LISTW) FIELDS(#OV_TEXT) TYPE(*WORKING
*****);
INZ_LIST NAMED(#OV_LISTD) NUM_ENTRIES(100) WITH_MODE(*CF
MESSAGE MSGTXT('Type in lines of text to be edited');
REQUEST BROWSELIST(#OV_LISTD);
SELECTLIST NAMED(#OV_LISTD) GET_ENTRIES(*NOTNULL);
ADD_ENTRY TO_LIST(#OV_LISTW);
ENDSELECT;
USE BUILTIN(TRANSFORM_LIST) WITH_ARGS(#OV_LISTW #OV_TEM
USE BUILTIN(OV_SYSTEM_SERVICE) WITH_ARGS(START #OV_EDIT

```

The following sample RDML function requests that you specify a program (.EXE) name and any parameters to be passed to it. An attempt is then made to start the specified program executing. You may start the application by using the START or the START_UNIQUE option thus allowing you to see the differences between these options when starting a program that is already active:

```

FUNCTION OPTIONS(*DIRECT);
DEFINE FIELD(#OV_PGM) TYPE(*CHAR) LENGTH(50) LABEL('Program
DEFINE FIELD(#OV_PARMS) TYPE(*CHAR) LENGTH(50) LABEL('Parame
DEFINE FIELD(#OV_RETC) TYPE(*CHAR) LENGTH(2) LABEL('Return (
DEFINE FIELD(#OV_ERRN) TYPE(*DEC) LENGTH(7) DECIMALS(0) LA
DEFINE FIELD(#OV_REQUEST) TYPE(*CHAR) LENGTH(20) LABEL('Sta
*****);
BEGIN_LOOP;
REQUEST FIELDS(#OV_REQUEST #OV_PGM #OV_PARMS (#OV_RETC
BEGINCHECK;
VALUECHECK FIELD(#OV_REQUEST) WITH_LIST(START START_UNIC
ENDCHECK;
USE BUILTIN(OV_SYSTEM_SERVICE) WITH_ARGS(#OV_REQUEST #O'

```

```
IF COND('#OV_RETC = OK');  
MESSAGE MSGTXT('Program started or it is already started');  
ELSE;  
MESSAGE MSGTXT('Error detected when attempting to start program');  
ENDIF;  
END_LOOP;
```