LANSA Application Design Guide

- About this Guide
- Object Naming Standards
- The Data Dictionary
- The Database
- Processes and Functions
- Producing Reports
- Pop-up Windows
- Prompt Key Processing
- Processing Arrays
- Action Bars and Pull Down Menus
- Examples of using LANSA
- Application Performance on IBM i
- IBM i SAA/CUA Implementation
- Generated Code C and SQL/ODBC Considerations
- Portability Specifics
- LANSA Limits
- Unicode and Internationalization
- Client/Server Applications
- Icons and Bitmaps
- Create Your Own Built-In Functions
- Application Templates

Edition Date October 22, 2014 © LANSA

About this Guide

The LANSA Application Design Guide has been written for both experts and novices whether creating or maintaining LANSA applications.

This guide should be used:

- Before commencing on the design of any LANSA based application.
- By new LANSA users in order to become familiar with naming standards and operating system and database constraints.
- As a way of triggering new approaches to designing and implementing systems.

This guide is not a set of standards. The examples given should be used as suggestions rather than the only way of doing something. The examples and suggestions have been mostly derived from the experiences of existing LANSA users.

The contents have been arranged in three broad categories:

- Chapters 1 to 8 provide information that is relevant to designing applications for all platforms supported by LANSA. Wherever information is platform-specific, it is indicated as "Platform Considerations".
- Chapters 9 to 12 contains information specific to designing RDML applications for the IBM i.
- Chapters 13 to 16 provide information specific to designing RDMLX applications for all platforms. Some information may be irrelevant for the IBM i.
- Chapter 19 describes how to build your own Built-In Functions.
- Chapter 20 gives an overview of Application Templates. It describes what they are and how you can either use the templates supplied or create your own. Application Templates save developer time as well as help to maintain site standards. It is well worth your while investigating what Application Templates have to offer.

1. Object Naming Standards

Basic Object Naming Rules

Use naming standards that only use letters of the English alphabet (A - Z) and numbers (0 - 9) and underscore (_) in all newly designed applications. The use of other characters should be avoided at all times. Although this is not a mandatory requirement it will mean that objects do not have to be automatically renamed if built on a non-IBM i platform.

Also see

LANSA Object Names in the *Technical Reference Guide* for object naming conventions and reserved words.

1.1 Package Development and Naming Standards

1.1 Package Development and Naming Standards

The term "Package Development" refers to applications developed specifically for sale on the open market.

All development, but especially package development, presents a particularly crucial problem with regard to naming standards. This is the problem of duplicated names.

For example, what happens if you design a package with a customer master file called CUSTMST, install it onto a new user's system and then find that the user has an existing customer master file called CUSTMST?

The best method of avoiding this situation is to set - and strictly follow - naming standards.

The suggested standard is to prefix all objects with two characters to identify the package. For example OE (order entry package) or GL (general ledger package).

2. The Data Dictionary

To achieve the full benefits of the Data Dictionary in LANSA's centralized Repository, certain standards need to be adopted and rigorously enforced. These include:

- Establishing and controlling who is authorized to add and update details in the dictionary.
- Ensuring elements and fields are defined only once.
- Whoever is authorized to maintain the data dictionary details should be committed to the concept of using a corporate-wide vision. Every element should be considered for it's impact on the whole organization, not just for one program in one application.

All designers and application developers should refer to the data dictionary for the definition of an element.

Using a centralized Repository may extend the design phase of a project but this will easily outweigh the benefits achieved during the development and maintenance phases.

Consistency in validity checking, error messages and the names used on screens and reports and provides immeasurable benefits to end users.

2.1 Field Naming Conventions

A suggested naming standard for fields in the data dictionary is:

For In-House Development For Package Development

XXXXXXiii PPXXXXiii

Where the following points apply:

- No field name should be longer than 9 characters.
- No field name should contain a "#" (hash/number) symbol or an "@" (at) symbol.

The XXXXXX / PPXXXX part of the name should be from 1 to 6 characters in length and as meaningful as possible given the size limitations. For example:

In-House Development

PARTNO	Part number
CUSTNO	Customer number
ITEMNO	Item number
SALQTY	Sales quantity
SALAMT	Sales amount
SALAVG	Sales average

Package Development

	-
OEPTNO	Part number
OECUST	Customer number
OEITEM	Item number
MKSAQT	Sales quantity
MKSAAM	Sales amount

OEPTNO, OECUST, and OEITEM are order entry packages, and MKSAQT, MKSAAM, and MKSAVG are marketing packages.

• "iii" is optional and should only be used for "working" or "temporary" versions of the original XXXXXX / PPXXXX field in RDML programs. The iii suffix should be standardized and only a finite number of variations allowed. For example:

NXT	Next value of the field
PRV	Previous value of the field
WRK	Working value of the field
USE	Exchanged/using value of the field
RN1	First rename of the field
RQS	Requested value of the field
VIR	Virtual field definition
TOT	Totaled value of field

- File definitions should only ever use the XXXXXX or PPXXXXversion of the field name (with the possible exception of virtual fields). The versions with the iii suffix should be restricted for use as "working" fields within RDML programs.
- All "iii" suffix versions of the field should be defined in the dictionary. No RDML program should specifically define such fields, even when the correct REFFLD (refer field) option is used. This greatly increases the cross-referencing power of LANSA and standardizes program work variable naming conventions.
- In a "package" environment the PP component of the name should be standardized. For example:

GL	General ledger
AR	Accounts receivable
OE	Order entry

One advantage of this type of naming standard is that when an attempt is made to change the dictionary and field name XXXXXX or PPXXXX is specified, all the iii variations will also appear due to LANSA's generic search and display facility. This serves as a continual prompt to remember to change/check all the versions of the same field.

The LANSA Product Support Department in Australia maintains a register of all known PP prefixes. If you are assigning a PP prefix it may be advisable to contact your product vendor to check that the prefix has not already been used, and to have your prefix added to the register to prevent others from using it.

2.2 Some Dictionary Guidelines

Some suggested guidelines for the definition of fields stored in the data dictionary are as follows:

- Dictionary maintenance should be assigned as a task to only one person within a project team, or even to just one person within an organization.
- Remember the concept of the corporate dictionary when specifying any field attribute. You are setting and standardizing a "corporate" definition of the field. When defining descriptions, column headings, labels, attributes or edit details this is particularly important. Try to forget the image of this field as it will appear on a specific screen or report.
- Fields must be named as per the previous section and all "iii" variations of the field must be defined in the dictionary.
- Use alphanumeric fields whenever possible. Use the rule "if you don't want to add it up it should be alphanumeric". LANSA provides many facilities which work more easily on alphanumeric fields (e.g.: generic searching).
- All packed decimal fields should be of odd length.
- Descriptions for all fields must be provided and should make use of both upper and lowercase characters. The description should apply at a corporate level rather than at a specific application level. Use CUA standards for word casing and phrasing whenever possible. Avoid making descriptions too long.
- Column headings should be precise, use upper and lowercase characters , and not be significantly wider than the field itself. This avoids wasting space on reports:

should	l be	Part	
Part Number Status	====	=====>	Status
XXX		XXX	
XXX		XXX	
XXX		XXX	

• Only one or two lines of the column heading should be used. For example:

Part	Customer	
Number	Number	Date Due

• Field labels should always be specified. Upper and lower case characters

should be used.

Part Number Cust Number Date Due

• Input attributes should not normally be specified. The system default attributes should be correct for most fields.

In SAA/CUA compliant partitions the following input attributes should always be specified (in addition to any system default attributes):

- PBEN Panel body normal input field. For normal or non-significant fields (e.g. Zip Code).
- PBEE Panel body emphasized input field. For important, significant or key fields (e.g. Customer Number).
- Output attributes should not normally be specified. The system default attributes should be correct for most fields.

In SAA/CUA compliant partitions the following output attributes should always be specified (in addition to any system default attributes):

- PBCN Panel body normal output field. For normal or nonsignificant fields (e.g. Zip Code).
- PBCE Panel body emphasized output field. For important, significant or key fields (e.g. Customer Number).
- An edit code or edit word should be provided for every numeric field defined in the dictionary. A suggested basic standard for edit code usage is:
- Y For any form of date field.
- 4 For numeric fields that will never be negative.

M For numeric fields that may be negative.

• HELP text must be input for all fields defined in the dictionary.

A standard HELP text form should be set up and used as the basis for all field level HELP text input into the dictionary. Use of attributes and \$\$

substitution variables can vastly reduce the amount of keying required to input a field's HELP text. (Refer to Using Substitution/Control Values in HELP Text in the LANSA for iSeries User Guide and Substitution/Control Values in the Technical Reference Guide for more details.)

Using a standard layout for all HELP text produces a consistency across your entire application. End users will become familiar and comfortable with the layout used.

• Any field that appears on a screen or a report should be defined in the data dictionary, rather than in the specific RDML function which displays the screen or produces the report.

This point should be emphasized to programmers used to "external" field definitions only being available when they use a file actually containing the field. In LANSA any field defined in the data dictionary can be used in any RDML function.

Some of the advantages of using the LANSA data dictionary for more than just defining the fields (or elements) which make up the records of files in the database are:

- Less repetition of "work" fields. After a period of time most required work fields will already be in the data dictionary.
- More comprehensive cross-referencing capabilities.
- Fields in the dictionary can have HELP text. Those defined in programs cannot. This is particularly important for fields intended for display.

2.3 Portability Considerations

Data Dictionary Definitions

When working with fields, you should be aware of the following:

- The following field attributes are supported: ND LC VN AB RZ M10 M11 FE RA PBIN PBFP PBBR PBCM PBGH PBNT PBET PBEN PBEE PBCN PBCE PBSI SBIN
- Numeric fields that may contain a negative value and are presented on user interface panels should always have an edit code or edit word specified. e.g. IBM i presents -1 as "J" but X_UIM (Visual LANSA User Interface Manager) presents it as "1"
- Numeric fields that have a decimal component and are presented on user interface panels should always have an edit code or edit word specified such that a decimal point is always shown.
- A complex logic check that calls a 3GL program is not supported.
- Some system variables have fixed values because they have no real meaning outside of an IBM i context. For example:

*JOBD is always QBATCH

*CPU_NUMBER is always 0

*GUIDEVICE is always Y

*OUTQNAME is always QPRINT

*OUTQLIB is always QGPL

*MSGQNAME is always the job name

*MSGQLIB is always QGPL

*GROUP_OWNER is always *USRPRF.

- Some system variables have new values. For example*CPUTYPE can vary depending on the platform.
- When working with fields on files that have the attribute *Convert special characters in field names* set to **Yes**, you should be aware of the following when table columns are created from the field names:
 - Special characters in names (i.e., characters that are not A Z, 0 9 or _)

are replaced according to position. For example: a field named DC@FLD would be renamed DC_FLD; a field named @FLD would be renamed XFLD.

- Names that are recognized as SQL keywords will have their second character changed to an underscore. For example: a field named SECTION would be renamed S_CTION.
- Avoid naming any new objects with a name that conflicts with an SQL keyword. Again this prevents automatic renaming. Refer to SQL/ODBC Grammar:Keywords in Generated Code C and SQL/ODBC Considerations.

3. The Database

- 3.1 File Naming Conventions
- 3.2 Some Database Guidelines
- 3.3 Portability Considerations

3.1 File Naming Conventions

A suggested naming standard for files in the data dictionary is:

For In-House Development For Package Development

Physical: XXXXXX	PPXXXX
Logical: XXXXXXnn	PPXXXXnn

Where these points apply:

- File names should be no longer than eight characters.
- File names should **not** contain a "#" (hash/number) symbol or an "@" (at) symbol.
- The XXXXXX / PPXXXX part of the physical or logical file name should be exactly six characters in length and as meaningful as possible, given the size limitations.

The nn portion should be a two digit number assigned sequentially to each file. For example:

CUSTMRCustomer filePRODCTProduct master filePRODCT01Products in category orderPRODCT02Products in class/category orderITEMLNItem lines fileSALESHSales history fileSALESH01Sales history by customerSALESH02Sales history by product

In house development:

SALESH02Sales history by productSALESH03Sales history by month/customer

Package development:

OECSMF	Customer file
OEPRMF	Product master file
OEPRMF01	Products in category order
OEPRMF02	Products in class/category order
OEITLN	Item lines file
MKSHF	Sales history file
MKSHF01	Sales history by customer
MKSHF02	Sales history by product
MKSHF03	Sales history by month/customer

Note: Avoid the trap of trying to use meaningful logical file names (i.e., attempting to make the name reflect the order of the keys in some way). This is practically impossible to achieve consistently across any reasonable number of file definitions.

We recommend the method of just numbering the files sequentially, as at least this ensures that logical file names reflect the physical files on which they are based. If you wish, put the description of the key order into the 40 byte file description where it will be visible to users at most times.

3.2 Some Database Guidelines

Some suggested guidelines for the definition of database files are as follows:

- File definition maintenance should be assigned as a task to only one person within a project team, or to even just one person within an organization.
- All files (physical and logical) should have an accurate and concise description which uses upper and lower case characters.
- Avoid the use of too many logical views in applications to be deployed on the IBM i. The maintenance of logical file access paths (indexes) is probably the largest resource user in any IBM i.
- For any logical view that is rarely used, consider using the RDML OPEN command with the USE_OPTION(*OPNQRYF) and KEYFLD parameters if the application is only to be deployed on an IBM i. This will achieve exactly the same result without causing the overhead of continual maintenance of another access path.
- Investigate the use of the dynamic select option on logical files using the select/omit criteria. If the required file has the same key(s) as an existing logical file and selects a large percentage of the records in the file, the "dynamic select" option is a good solution.

Alternatively use the RDML OPEN command with the USE_OPTION(*OPNQRYF), QRYSLT and KEYFLD parameters if the application is only to be deployed on an IBM i. This is a viable alternative, especially if the order specified in the KEYFLD parameter is identical to that in an existing logical file.

- Use the IMMED (immediate) maintenance option on a logical file with care.
- Use as many file level validation checks as possible. There are not many situations in which a validation check cannot be performed by one of the six standard file level validation checks.
- There are many advantages in using file level validation checks, including:
- Most RDML programs can be written without any online validation.
- The rules can be changed without modifying or re-compiling application programs which use the file.
- The rules are standardized and can be tested by the database designer before one line of RDML code is written.

- The rules protect the database, even from a rogue RDML program.
- The error messages are standardized and consistent.
- Avoid the use of multi-membered files. Even though LANSA supports access to multi-membered database files, their use is not recommended as they are prone to become a maintenance problem. Either use another file altogether, or include an extra key into the file definition.

Additionally, the direct portability of multi-membered files to SQL based systems is not possible.

- Investigate the use of database triggers to make the database "smart" rather than trying to make each individual function "smart".
- When LANSA creates database files in IBM i, unless otherwise specified, it will automatically make the record format name exactly the same as the file name. If you intend to write RPG application programs which will access database files set up by LANSA, you have two options:
- Before making the file definition operational, take the option from the File Definition Menu (on either file create or review) to change the file's record format name.
- Rename the file record format within the RPG program itself.

3.3 Portability Considerations

Database Definitions

When working with files, you should be aware of the following:

- 10 character names for libraries will be truncated on the left to 8 characters. We recommend that you avoid this truncation by having library names that are 8 characters or less.
- Special characters in names (i.e. characters that are not A-Z, 0-9 or _) are replaced according to position. For example:
 - A library named DEMPGMLIB would be renamed EMPGMLIB
 - A file named DC@FILE would be renamed DC_FILE
 - A file named @FILE would be renamed XFILE.
- Batch control files must be defined in the same library as the base physical file.
- Extended virtual code type Substringing using *RECORD is not supported.
- Complex logic check that calls a 3GL program is not supported.
- 3GL virtual code must be specified in ANSI C
- Select/omit criteria cannot span lines on a LANSA/AD screen panel for select/omit criteria. This means that each individual select/omit condition must be totally contained within 29 characters when using Visual LANSA. For example - COMP(LE 'XXXXXXXX') is valid COMP(LE 'XXXXXXXXXXXXXXXXXXX') is invalid.
- The following select/omit criteria keywords are supported COMP, CMP, RANGE and VALUES. The ALL keyword cannot be supported. Attempting to compile a file with the ALL keyword will result in an error in the file generation.
- The emulation facilities for IBM i select/omit criteria are only provided to support the **reading** of selected information from database tables. They are not provided to (or designed to) allow you to control what information is inserted or updated into database table rows. If you are using an application design that relies on the use of uniquely keyed logical files (with associated select/omit criteria) to control the **content** of the your database then you may have to revise this component of your design.
- Collating sequence for ASCII is different from EBCDIC refer to Generated

Code C and SQL Considerations.

• Commitment control may behave differently - refer to refer to Generated Code C and SQL Considerations.

4. Processes and Functions

- 4.1 Before Coding RDML Processes and Functions
- 4.2 Process and Function Naming Conventions
- 4.3 Break Business Functions Down into Processes and Functions
- 4.4 Build Menu Structures
- 4.5 Use Application Templates
- 4.6 Use an Exchange List and the OPTION(*ALWAYS) Exchange Option
- 4.7 A Message for every Situation
- 4.8 Top Down and Structured Programming Techniques
- 4.9 Portability Considerations

4.1 Before Coding RDML Processes and Functions

These guidelines are presented in the form of a checklist as every item should be satisfied before a single RDML process or function is coded.

- All data dictionary naming standards and guidelines are known by/available to every member of the project team.
- All file definition, naming standards and guidelines are known by/available to every member of the project team.
- All process and function naming standards and guidelines are known by/available to every member of the project team.
- All known data dictionary entries, validation rules and HELP text have been input.
- All known files in the database have been designed, implemented and are understood by all members of the project team. Additionally, all file level validation checks have been implemented and tested.
- Preferably the entire database should be "visualized" in the form of a relational map. Updated versions of this map should be given to all members of the project team.
- The entire application system has been broken down into identifiable processes and functions which will perform all the required business functions (i.e., the whole system is designed before coding is commenced).
- The complete backup and recovery strategy to be used by the application system has been designed.
- All known system variables have been identified, implemented and tested. All members of the project team are aware of their existence and what they can be used for.
- A global exchange list has been designed for all programs in the application system. All members of the project team are aware of its existence and how and why it is used.
- Application Templates are available to automatically generate most common types of online transactions and report production programs. These have been tested and conform to all site standards for "look" and "feel".

4.2 Process and Function Naming Conventions

All the naming standards suggest a standard prefix for packaged applications (expressed as PP). For example OE (order entry package) or GL (general ledger package).

Following is a naming standard recommended for processes and functions:

For In-House Development For Package Development

Ppaaaooo

Processes: Oooaaa

Functions: Ooonnnn Ooonnnn

Where:

ooo indicates the basic or primary object that the process uses. For example:

CUS	Customers
ORD	Orders
SYS	General system procedures
LGR	Ledgers

aaa indicates the type of actions taken on the object by functions within the process. For example:

MNT	Maintenance
REP	Reports
WRK	Work with
CTL	Control routines
SBR	General subroutines
SEL	Selection subroutines
MNU	Displays a menu

nnnn is a sequentially assigned number within the "ooo" grouping. For example:

ORD0001	Enter new order details
ORD0002	Modify current order details
ORD0003	Modify back order details

- Every process name within a partition must be unique. This standard is enforced by LANSA.
- Every function name must be unique within a process. Function names may be identical if they belong to different processes.
- No process should contain more than 20 functions (unless is a special subroutine or prompt key "grouping" process).

4.3 Break Business Functions Down into Processes and Functions

The task of breaking a set of business functions down into processes and functions is one of the hardest steps. These guidelines may assist:

4.3.1 For Traditional Application Systems

4.3.2 For Object/Action Orientated Designs (Action Bar Systems)

4.3.1 For Traditional Application Systems

Given that the database is usually designed before the applications are created, you should first examine it closely and attempt to identify and define primary "objects".

For example, a database being used to store corporate order details may contain the following objects:

Customers

Orders

Products

Assign a unique 3 letter identifier (as per the guidelines previously discussed) to each identifiable object stored in the database. For example:

Customers CUS

Orders ORD

Products PRO

Next, examine each object, and decide what "actions" must be performed to satisfy all business requirements. To each "action" (now called a FUNCTION) assign a unique identifier as recommended:

Customers (CUS)

maintain	CUS0001
show outstanding orders	CUS0002
print all	CUS0003
print by state	CUS0004

Orders (ORD)

maintain	ORD0001
print	ORD0002
archive	ORD0003

Products (PRO)

maintain	PRO0001
print details	PRO0002
print sales analysis	PRO0003

Finally, group the functions, which work on each object type into broad logical "categories". To each "category" (now called a PROCESS) assign a unique identifier as per the standards previously discussed:

Customers (CUS)

Work with customers (CUSWRK)	maintain	CUS0001
	show outstanding orders	CUS0002
Produce customer reports (CUSREP)	print all	CUS0003
	print by state	CUS0004

Orders (ORD)

Work with orders (ORDWRK)	maintain	ORD0001
	print	ORD0002
	archive	ORD0003

Products (PRO)

Work with products (PROWRK)	maintain	PRO0001
Produce product reports (PROREP)	print details	PRO0002

Note that the assignment of functions to categories (i.e., processes) is completely arbitrary, and may even seem inconsistent at this stage. For instance both customer and product reports have been placed into separate processes, but order reports have not.

The most common mistake made at this stage is to consider that a process is merely a menu. It may certainly be used as a menu, and in many instances works just fine as a menu, but its most important use at this stage is its ability to logically "group" or "categorize" functions together.

4.3.2 For Object/Action Orientated Designs (Action Bar Systems)

If you plan to design an action bar driven application system you should use a different technique to break the business requirements down into processes and functions.

The recommended technique is very well described in the IBM manual *Common User Access: Basic Interface Design Guide* (SC26-4583). An appendix in this manual titled "Designing an Object-Action Orientated Application" describes a method for this breakdown process.

An "Object-Action" approach is essential to designing and implementing action bar driven applications.

Read the complete text of *Action Bars and Pull Down Menu Applications* before attempting to design or implement an action bar driven system. With the prototyping details in that chapter and the break down procedures from the IBM CUA manual you will find that it is very easy to design, prototype and then implement an action bar driven system.

It should be far easier, and produce more consistent results, than a more traditional "Action/Object" design approach would.

4.4 Build Menu Structures

This section describes the design and building of menu structures for traditional application systems. If you are designing an action bar driven system you should still read this section (from a different perspective), because action bars are often initially invoked from menus.

Once the entire application system has been broken down into a series of processes and functions it is possible to consider how its online components will be fitted together into a menu structure (assuming that you intend to use LANSA to build your menus).

Using the example application system described in the previous section, we can visualize it as a series of independently accessible menus as follows:



However, taking it one step further, by using the "process attachment" capability of LANSA (described in Other Processes on the Process Menu in the *iSeries User Guide*), we can quickly create a generalized "menu" (actually a new process called SYSMNU) for end users of this system. This could be visualized like this:





One step further still, and a "tailored menu" can be created, say, for the accounting department. This "tailored menu" is actually another process called ACCMNU, and it can be visualized as follows:



The MAIN POINT of the above exercise is that the grouping of functions into processes at design time need not have anything at all to do with their final organization within the complete application system.

4.5 Use Application Templates

It is recommended that all RDML programs be created from an "Application Template".

By using "Application Templates", it is easy to ensure that site standards for program layout, screen layout, report layout and transaction styles are closely adhered to.

That is, a template can control and pre-define the "look", but more importantly the "feel", of an application.

Some things to consider about templates are:

- They can generate complete programs when "Creating a new function".
- They can generate just small blocks of code directly from the RDML editor (e.g.: your site standard report header layout).
- They increase accuracy, productivity and consistency.
- They can standardize the "feel" or "style" of any number of standard transactions to exactly your site standards. This is more important to end users than the "look".

It is strongly recommended that you put aside time to develop or enhance the shipped application templates to exactly match your site standards.

This process is time consuming and requires a lot of testing.

The short term benefits may not even be apparent.

However, the long term benefits will justify the cost of doing this.

A recommended technique for doing this is "retro-fitting".

Create an RDML application program that performs some common type of "action" on, say, a customer master file.

Work the function over and over, until it represents exactly what you like in terms of a site standard for the "look" and "feel" of the "action".

Copy this function into a template, then "generalize" it so that it is capable of generating functions that perform the "action" on any file or files.

You now have a standard for the "look" and "feel" of that type of "action".

The consistency that this template produces will be of vast benefit to your end users and make the creation of programs that need to perform the "action" very fast, very easy and very accurate.

Refer to Application Templates for more information.

4.6 Use an Exchange List and the OPTION(*ALWAYS) Exchange Option

The EXCHANGE LIST is the preferred way of passing information from function to function within LANSA applications. Only ever use PARAMETERS to pass information from RPGIII or CL into LANSA applications.

When a new application system is being defined, it is useful to identify any information which should always be available to be exchanged by every program within the application. This can save having to make later complex amendments to program suites, AND will often improve application performance by saving repetitive logic in many different programs.

Let's take an example of information falling into this category within an application suite - a general ledger system. During the design phase of this system the analyst or designer would fairly quickly identify that:

- Nearly every program needs to know the company number of accounts being worked with.
- Most online programs display the description of the company being worked with on line 2 of any screen.
- Many programs request the user input an account number. It would be useful if the last account number being used was always presented as a default value.

The best way to achieve these aims is to define a "global" exchange list as in the example below. Such a list should be included into all programs which are part of the application system - whether these programs appear to use the information or not.

EXCHANGE FIELDS(#COMPNOUSE #COMPDSUSE #ACCNOUSE) OPTION(*ALWAYS)

Some points to note about this command are:

• The use of the names #COMPNOUSE, #COMPDSUSE and #ACCNOUSE. According to the standards of field naming recommended by this guide, these fields are clearly work fields. The actual database fields would be called #COMPNO, #COMPDS and #ACCNO.

The use of work fields is done for one reason only. To prevent an RDML function that is FETCHing or SELECTing data from the database (i.e., fields named #COMPNO, #COMPDS and #ACCNO) from inadvertently

overwriting the global exchange values with something that may well be rubbish. The three 'USE' versions of the fields would all be defined in the data dictionary.

- The requirement that this command be included into every program in the suite can be more rigorously enforced if the command is included into any application templates used to generate this system.
- Any template used by programs in this suite should contain the global EXCHANGE command. However, it may also be advantageous to include standard layouts for screens or reports with standard locations for the globally exchanged information.

For instance, the following sample GROUP_BY (used for screens layouts) and DEF_HEAD (used for report layouts) commands might be included into a template:

GROUP_BY NAME(#XXXXXXX) FIELDS((#COMPNOUSE *L3 *P (#COMPDSUSE *L3 *P20 *NOID))

DEF_HEAD NAME(#XXXXXXX) FIELDS((#COMPNOUSE *L1 *P (#COMPDSUSE *L1 *P20 *NOID) (#TIMEDATE *L2 *P1 *LABEL) + (#JOB *L2 *P20 *LABEL)(#USER *L2 *P45 *NOID) + (#JOBNBR *L2 *P52 *NOID))

4.7 A Message for every Situation

The LANSA RDML language is very good at issuing and handling messages. Three different types of messages can be issued by any RDML function via the MESSAGE command. These are:

- An information message (type *INFO). The message is issued and the program continues executing, but the message does not appear until the next DISPLAY or REQUEST command is processed (even if it is not in the program which issued the message).
- A status message (type *STATUS). The message is issued, appears immediately on the bottom of the screen (no matter which program presented the screen), and the program continues execution. When the next DISPLAY or REQUEST command is processed the message disappears.
- A window message (type *WINDOW). The message is issued immediately and appears in a "window" which overlays what is currently on the screen (no matter what program presented the screen). The program then stops executing and waits for the user to press the enter key. A reply may be entered to the message. The reply is then made available for processing by the RDML program.

Significant productivity and design benefits can be achieved by implementing the standardized use of messages in all sorts of situations. Review the examples provided.

Example 1: Information Messages - Instructions to the User

Example 2: Status Messages - Keeping the User Informed

Example 3: Window Messages - Confirming User Actions

Example 4: Information Messages - Batch Programs

Example 1: Information Messages - Instructions to the User

Consider the following very simple data entry program:

GROUP_BY NAME(#CUSTOMER) FIELDS(#NUMBER #NAME #A BEGIN_LOOP REQUEST FIELDS(#CUSTOMER) INSERT FIELDS(#CUSTOMER) TO_FILE(CUSMST) CHANGE FIELDS(#CUSTOMER) TO(*NULL) END_LOOP The automatically designed data entry screen would probably look something like this:

Number	 -	
Name	 	_
Address	 	
Zip code		

The programmer may then use the screen design facility to modify the screen to include "instructions" to the user, such as:

Input customer details and press enter

Number <u>.</u>	-	
Name		
Address		
Zip code		
However, a faster and more consistent alternative to the continual insertion of "instructions" on LANSA designed screens is to use the MESSAGE command. For example, if the RDML program was altered as follows:

> GROUP_BY NAME(#CUSTOMER) FIELDS(#NUMBER #NAME #A MESSAGE MSGTXT('Input customer details and press enter') BEGIN_LOOP REQUEST FIELDS(#CUSTOMER) INSERT FIELDS(#CUSTOMER) TO_FILE(CUSMST) CHANGE FIELDS(#CUSTOMER) TO(*NULL) MESSAGE MSGTXT('Customer accepted ... enter next customer') END_LOOP

then the initial screen presented to the user would look like this:

Number	
Name	
Address	

Zip code \dots	
------------------	--

Enter customer details and press enter

After details of the first customer have been successfully input, the screen is effectively cleared and a message indicating that the "customer was accepted and that the next customer's details should be input" will appear.

There are some significant advantages in using messages for user instructions. These include:

• Less time spent by programmers arbitrarily inserting instructions on the

screen.

- Consistency in all applications.
- Instructions (which are usually superfluous to experienced end users) will always appear on the message line.
- The ability to issue multiple lines of instructions by executing multiple MESSAGE commands.
- The message is generally more conversational (or at least easier to make conversational) than a piece of static instruction on a screen.
- If a message from a message file is used (rather than just text as in the previous example), then the user has the option to place the cursor on the "instruction" message and press HELP.

The second level text of the message will then appear. This facility allows a full page of "extended instructions" to be displayed to the user.

Overall, there are not many situations in which the "instructions" which are so often placed onto screen formats cannot be replaced by messages.

Example 2: Status Messages - Keeping the User Informed

Status messages can add to the "user friendliness" of any system by keeping the user informed of what is going on. Some examples of where they could be used are:

• In an online report production program.

For instance, the user requests that a particular customer report is to be produced (online). It is a nice touch if a message such as ."Customer report being produced please wait" immediately appears.

• In any online task that takes an extended time.

For instance, processing and validating a large batch of general ledger transactions online can be made more "friendly" if a message such as the following one appears after every 20 transactions are processed "nn G/L transactions processed. nn accepted. nn rejected" It reminds the user that the computer is working for them.

Example 3: Window Messages - Confirming User Actions

Very often, it is necessary to get an end user to confirm a request. Typically, this will involve things like deletion confirmation, or confirming that a tape is mounted, or that a large batch report run should really be submitted, etc.

Usually a special purpose built screen can be displayed for user confirmation.

However, it is more productive - and often more aesthetic - to use a "message window".

The following are examples of overlaid message windows which can be easily produced by the MESSAGE command:

Number A627478 Name ACME ENGINEERING CORPORATION Address . . . 121 SMITH STREET, ANYTOWN. Zip code . . . 18475

.....

: Confirm this customer is to be deleted (Y or N) :

: Reply : _ :

:.....:

Enter number of action required : 3

- 1. Power down the system
- 2. Perform weekly backup procedures
- 3. Perform daily backup procedures

.....

: Confirm daily backup tape is mounted (YES or NO) :

: Reply : ____ :

......

Produce Customer Summary Report

Print customers in state C

Print Zip code range 34859 to 48579

Print only those with orders . . . Y

.....

: Confirm Customer Summary Report should be run (Y N):

: Reply : _ :

:.....:

A message window has the advantage that it overlays the current display (no matter what program originally presented the screen) and can be positioned to overlay the top, middle or bottom of the screen. Thus you can ensure that the user can see just what they did to cause the window to appear.

Example 4: Information Messages - Batch Programs

Most programmers quickly realize that the message handling facilities within LANSA can improve the appearance and friendliness of online applications, as well as reducing programming time and maintenance costs.

LANSA messages can also provide significant benefits in batch programs - particularly those that perform data validation and/or produce reports.

Consider the following section of a batch program for processing name and address details which have arrived on the system via magnetic tape:

DEFINE FIELD(#ERRTXT) TYPE(*CHAR) LENGTH(100) LABEL('Errc DEF_LINE NAME(#NAME) FIELDS(#CUSTNO #ADDRESS1 #ADDRE DEF_LINE NAME(#ERROR) FIELDS(#ERRTXT) IDENTIFY(*LABEL)

SELECT FIELDS(#NAME) FROM_FILE(TAPEINP) PRINT LINE(#NAME) INSERT FIELDS(#NAME) TO_FILE(NAMES) VAL_ERROR(*NEXT) IF_STATUS IS_NOT(*OKAY) MESSAGE MSGTXT('Details of this name not added to database') 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

The above program reads all name and address details from a name and address tape input file called TAPEINP. Each name and address processed is printed onto a report in a line called #NAMES.

The program then attempts to add each name to a name and address file called NAMES. File NAMES has been set up with many dictionary or file level validation checks, so the INSERT command will fail if they are not met.

When such a validation error occurs, a general purpose message will be issued indicating that the name and address just printed onto the report was NOT added

to the database file NAMES.

Then the Built-In Function GET_MESSAGE will "read" all messages from the program message queue and print the text in a line called #ERROR. All the data dictionary and file level validation errors will be printed before the final message issued by this program is printed.

By using the MESSAGE command and the GET_MESSAGE Built In Function together, it is possible to produce entire reports by issuing, and then "reprocessing", messages.

By using predefined messages from a message file and message substitution variables, it is possible to create and manipulate complex variable text strings for reporting purposes.

4.8 Top Down and Structured Programming Techniques

This guideline is very simple, yet it will allow LANSA programs to be completed more quickly and accurately.

When working on the code of a specific RDML function, always attempt to use a "top down" approach to coding. For instance, you may be given a program "specification" which looks like this:

- Repeat until program terminated at user request
 - Get the user to input an order number
 - Get details of the order, and handle record not found
 - Get details of all the order lines
 - Display the order details and the line details

You must now "translate" this specification into the LANSA rapid development language.

With a structured (or organized) approach to coding, the program will follow similar steps to the example below.

Note how the arrow --> is used to indicate the new lines inserted in each step:

Step 1

- --> ********* Do until terminated by EXIT or MENU key
- --> BEGIN_LOOP
- --> END_LOOP

Step 2

********* Do until terminated by EXIT or MENU key BEGIN_LOOP

--> ******** Request order number input until order is found --> ********* Build a list of all order lines

--> ******** Display the order and its lines to the user

END_LOOP

Step 3

```
--> ENDUNTIL
```

Step 4

```
--
> SELECT FIELDS(#LINES) FROM_FILE(ORDLIN) WITH_KEY(#ORD
--> ENDSELECT
```

```
--> ENDSELECT
```

******** Display the order and its lines to the user

END_LOOP

Step 5

Step 6

********* Do until terminated by EXIT or MENU key BEGIN_LOOP ********* Request order number until order is found DOUNTIL COND('#IO\$STS = OK')

Step 7

-

ENDUNTIL

********* Build a list of all order lines

CLR_LIST NAMED(#LINES)

SELECT FIELDS(#LINES) FROM_FILE(ORDLIN) WITH_KEY(#(ENDSELECT

********* Display the order and its lines to the user DISPLAY FIELDS(#HEADER) BROWSELIST(#LINES) END_LOOP

Step 8

```
REQUEST FIELDS(#ORDNUM)
  ******* Attempt to locate the order
           FIELDS(#HEAD) FROM_FILE(ORDMST) WITH_KEY(#(
  FETCH
 ********* Issue message if order was not found
--> IF STATUS WAS NOT(*OKAY)
--> ENDIF
 ENDUNTIL
  ********** Build a list of all order lines
 CLR_LIST NAMED(#LINES)
           FIELDS(#LINES) FROM_FILE(ORDLIN) WITH_KEY(#(
  SELECT
  ENDSELECT
  ******* Display the order and its lines to the user
 DISPLAY FIELDS(#HEADER) BROWSELIST(#LINES)
 END LOOP
```

Step 9

```
********** Do until terminated by EXIT or MENU key
 BEGIN LOOP
 ******** Request order number until order is found
            COND('#IO$STS = OK')
 DOUNTIL
 ********* Request user inputs an order number
            FIELDS(#ORDNUM)
 REQUEST
 ****** Attempt to locate the order
          FIELDS(#HEAD) FROM FILE(ORDMST) WITH KEY(#(
 FETCH
 ********* Issue message if order was not found
 IF STATUS WAS NOT(*OKAY)
--> MESSAGE
             MSGTXT('No details of this order found')
 ENDIF
 ENDUNTIL
 CLR LIST NAMED(#LINES)
          FIELDS(#LINES) FROM FILE(ORDLIN) WITH KEY(#(
 SELECT
 ENDSELECT
 ******* Display the order and its lines to the user
 DISPLAY
           FIELDS(#HEADER) BROWSELIST(#LINES)
 END LOOP
```

```
BEGIN LOOP
 ******** Request order number until order is found
            COND('#IO$STS = OK')
 DOUNTIL
 ******** Request user inputs an order number
            FIELDS(#ORDNUM)
 REQUEST
 ******* Attempt to locate the order
          FIELDS(#HEAD) FROM_FILE(ORDMST) WITH_KEY(#(
 FETCH
 ********* Issue message if order was not found
 IF_STATUS WAS_NOT(*OKAY)
            MSGTXT('No details of this order found')
 MESSAGE
 ENDIF
 ENDUNTIL
 ********* Build a list of all order lines
 CLR LIST NAMED(#LINES)
          FIELDS(#LINES) FROM_FILE(ORDLIN) WITH_KEY(#C
 SELECT
--> ADD_ENTRY TO_LIST(#LINES)
 ENDSELECT
 ******** Display the order and its lines to the user
           FIELDS(#HEADER) BROWSELIST(#LINES)
 DISPLAY
 END LOOP
```

You should find that well structured and documented programs are very easy to produce by using this type of approach, which basically consists of always:

- entering the structured programming constructs completely before coming back to fill in the details.
 For example: BEGIN_LOOP/END_LOOP, DOUNTIL/ENDUNTIL, SELECT/ENDSELECT, IF/ELSE/END, etc. etc.
- including the comments before the commands.

4.8.1 Paint Screen and Report Layouts Last

As IBM i 3GL compilers require that display and printer files exist before the corresponding program can be successfully compiled, 3GL programmers are in the habit of designing screens or reports first.

When using LANSA, significant productivity gains can result by NOT following this conventional IBM i 3GL technique. Let LANSA design the screen and report layouts as you code, and then you can modify the layouts when you know that the program works.

Some points about this technique are:

• When you need a screen to request or display information, think in terms of what you want to be on the screen, NOT where you want it on the screen.

For instance if you need the user to input a part number and a customer number, simply code the following and and then keep writing the RDML program logic:

REQUEST FIELDS(#PARTNO #CUSTNO)

What is initially important is that the user inputs the required information and that the program works correctly. The location of the information can be easily changed when the program is complete and functions correctly.

• The same idea applies to defining report lines. Think of what is to be printed on the line, NOT where it is to be printed.

For instance, if your report heading must include the date, time, company name and page number, then just code:

DEF_HEAD NAME(#HEAD01) FIELDS(#DATE #TIME #COMPANY #REI

..and keep on going. The detailed positioning of this information can be performed later when the rest of the program has been coded and tested.

- If this idea is used on a large scale, the development of an application system will proceed at a much faster pace. It is also possible to actually "prototype" systems at a very rapid pace if this approach is adopted.
- The final painting of screens and reports should be put off if possible until the program involved is completely tested.

- The final painting of screens and reports may be done by the system designer/analyst. An "end user" of the system may also be present to vet each layout in its final form.
- An additional benefit of using this technique is that the online Full Function Checker (which is relatively expensive to run) need only be run once per program (i.e., when the time comes to paint the final screen and report layouts).

4.9 Portability Considerations

Process and Function Definitions

When working with processes, you should be aware of the following:

- 10 character names for processes will be truncated on the left to 8 characters. For this reason we recommend that you avoid the left side truncation of process names by having process names that are 8 characters or less.
- Special characters in names (i.e., characters that are not A Z, 0 9 or _) are replaced according to position.

For example:

- A process named DEMPROC01 would be renamed EMPROC01
- A function named DC@FUNC would be renamed DC_FUNC
- A file named @FUNC would be renamed XFUNC.

and, interestingly:

- A process named CUSTPROC01 would be renamed STPROC01.
- A process named MUSTPROC01 would be renamed STPROC01.
- System commands can be attached to menus or action bars. However, these commands have meaning only to a specific operating system. This means that Windows commands have no meaning to IBM i just as IBM i commands will have no meaning in Windows.
- If you change the x_defppp.h file, all entry level processes should be recompiled
- DO NOT ignore warnings issued by the FFC (Full Function Checker) and code generators.
- If a partition has Display panel ids and Show date/time turned on, then for a given panel, if the panel id, date/time and panel title exceeds 60 characters the panel id will be dropped.
- Remember, functions are generated in C refer to Generated Code C and SQL Considerations.
- Message handling is different
- Review function naming standards in Generated Code C and SQL Considerations.

Message Handling

When working with message handling with Visual LANSA, you should be

aware of:

• Visual LANSA only supports message substitution variables of type *CHAR and *DEC. Other IBM i message types such as *HEX, *QTDCHAR and *BIN are not supported.

There are other IBM i internal message types such as *DTS, *SPP, etc that are not supported by Visual LANSA, just as they are not supported under IBM i in user defined messages.

- Visual LANSA is generally more forgiving and flexible than IBM i in handling substitution variables.
- Under IBM i every message substitution variable must be precisely defined and substituted in the message at execution time. This constraint is especially true of numeric values.

Rather than "cripple" Visual LANSA with the same very restrictive method, it was decided to leave it with a more flexible and open message substitution system.

The down side of this approach is that where Visual LANSA is building applications ultimately intended for IBM i systems, you must test message substitution variable processing under IBM i rather more rigorously than you have to under Visual LANSA.

- The "hiding" of packed decimal numeric message substitution values, inside what are to Visual LANSA alphanumeric fields, is not supported by Visual LANSA, in any way, shape or form.
- The use of IBM i special values, to control message text line formatting, etc., is not supported by Visual LANSA.

5. Producing Reports

- 5.1 Using Working Lists for Summary Reports
- 5.2 Producing Multiple Reports in One Pass over a File
- 5.3 Online and Batch Report Production
- 5.4 Writing More Flexible Reporting Programs Using the OPEN Command
- 5.5 Overriding Printer File Attributes using a Report Controller
- 5.6 Portability Considerations

5.1 Using Working Lists for Summary Reports

Where a "summary" style report is to be produced, and a relatively small number of summary records are to be printed (ie: less than 1000), then the LANSA object called a "working" list is a viable option.

In many situations, working lists save having to create additional access paths over database files (either directly via a logical file or by using the OPNQRYF option).

For example, consider the following instance where a "departmental expenditure" file called EXPEND is read:

FILE: "EXPEND"

Company	De	epartment	Amount
Number	Nu	ımber	Spent
#COMPN	0	#DEPNO	#AMOUNT
02	11	276.35	
01	14	100.12	
01	17	764.37	
02	12	1945.24	Ļ
01	19	89.12	
02	14	568.23	
01	15	375.89	

etc, etc

To produce a summary of expenditure by company, we could first define a "working list" called #SUMMARY as follows:

DEFINEFIELD(#SUMCOMPNO) REFFLD(#COMPNO)DEFINEFIELD(#SUMAMOUNT) REFFLD(#AMOUNT)

DEF_LIST NAME(#SUMMARY) FIELDS(#SUMCOMPNO #SUMAN TYPE(*WORKING) NBR_ENTRYS(50) This working list can be "visualized" as a multiple occurrence structure (or array):

```
      -------

      | Entry | Company | Amount |

      | Number | Number | Spent |

      |(implicit) |#SUMCOMPNO|#SUMAMOUNT|

      |------|-----|

      | 001 |
      |

      | 002 |
      |

      | 003 |
      |

      | 050 |
      |
```

Note that this allows for 50 different companies to be processed. If we attempted to process 51, the program would fail with a very specific error message indicating that the working list is full.

Now we can read all the "departmental expenditure" records and summarize them into the list called #SUMMARY:

```
SELECT FIELD(#COMPNO #AMOUNT) FROM_FILE(EXPEND)
LOC_ENTRY IN_LIST(#SUMMARY) WHERE('#SUMCOMPNO = #COMP
IF_STATUS IS(*OKAY)
CHANGE FIELD(#SUMAMOUNT) TO('#SUMAMOUNT + #AMOUNT
UPD_ENTRY IN_LIST(#SUMMARY)
ELSE
CHANGE FIELD(#SUMCOMPNO) TO(#COMPNO)
CHANGE FIELD(#SUMAMOUNT) TO(#AMOUNT)
ADD_ENTRY TO_LIST(#SUMMARY)
ENDIF
ENDSELECT
```

Note: The data can be read from file EXPEND in any order.

Next, we could sort the list to produce the summary report in company number order:

DEF_LINE NAME(#LINE01) FIELDS(#SUMCOMPNO #SUMAMOUNT)

SORT_LIST NAMED(#SUMMARY) BY_FIELDS(#SUMCOMPNO)

SELECTLIST NAMED(#SUMMARY) PRINT LINE(#LINE01) ENDSELECT

Finally, we could re-sort the list in descending order of amount spent, and produce another summary report (ie: companies ranked from highest spender to lowest spender):

DEF_LINE NAME(#LINE02) FIELDS(#SUMAMOUNT #SUMCOMPNO)

SORT_LIST NAMED#SUMMARY) BY_FIELDS((#SUMAMOUNT *DESC SELECTLIST NAMED(#SUMMARY) PRINT LINE(#LINE02) ENDSELECT

Some general notes about using working lists:

- Working lists are treated like database files in many ways, you can add data to them, update data already in them or clear them.
- The WHERE parameter can contain complex expressions. For instance the previous example could have used a parameter

WHERE('(#SUMCOMPNO = #COMPNO) *OR ((#SUMAMOUNT * 1.34) < #AMOUNT) *OR (#SUMCOMPNO = 99)').

This is a rather nonsensical expression, but it demonstrates the power of the LOC_ENTRY command.

• The SORT_LIST command can sort the list over more than one field, and can also support ascending or descending sorting.

5.2 Producing Multiple Reports in One Pass over a File

An often overlooked feature of LANSA is that up to EIGHT reports can be produced concurrently by any one program. Considering that often two or more programs are written to produce similar reports, (for example, one may be a detailed report, another may print selected details, and yet another may produce a summary), this is a significant feature.

If reports are usually produced together - and process basically the same information from the same files - then altering one of the programs to produce all reports concurrently can save both a lot of computer resource, AND simplify further maintenance of the reporting system.

5.3 Online and Batch Report Production

Report production programs usually perform a large number of database I/Os. As such they are usually unsuitable for online execution because they impact the performance of other online tasks too much.

This section describes some techniques which can be used to control when and where reports are produced.

The following generalized method for report production could be set up as an application template, and used to generate the initial version of any new report production programs:

*********** If this job is running online ********* _______ IF COND('*JOBMODE = I')********* Get user to specify mode and report parameters (#REPMODE + other report production parameters) REQUEST ******** Validate mode and any other run parameters here BEGINCHECK VALUECHECK FIELD(#REPMODE) WITH LIST(I B N) MSGTXT('Report mode must be I (inter), B (batch) or N (overnight)') **ENDCHECK** ******** Produce report or submit a batch job to do it OF FIELD(#MODE) CASE VALUE IS('= I') WHEN MESSAGE MSGTXT('Report xxxxxxx is being produced ... please wait') TYPE(*STATUS) EXECUTE SUBROUTINE(PRINT) WHEN VALUE IS(= B') PROCESS(*PROCESS) FUNCTION(*FUNCTION) JOBQ(*BA **SUBMIT** EXCHANGE(#XXXXXX #XXXXXX #XXXXXX #XXXXXX #XXXXXX USE **BUILTIN(CLR MESSAGES)** MESSAGE MSGTXT('Report xxxxxx submitted for batch production') WHEN VALUE IS('= N') PROCESS(*PROCESS) FUNCTION(*FUNCTION) JOBQ(*NI **SUBMIT** EXCHANGE(#XXXXXX #XXXXXX #XXXXXX #XXXXXX #XXXXXX USE **BUILTIN(CLR MESSAGES)** MSGTXT('Report xxxxxx submitted for overnight production MESSAGE

ENDCASE

********* Else if this job is running in batch

EXECUTE SUBROUTINE(PRINT)
ENDIF ******** =============================
********* Subroutine PRINT : Produce report

SUBROUTINE NAME(PRINT) *******
********* Define all print lines here

******** Produce report here

********* Close report and issue completion message

ENDPRINT
MESSAGE MSGTXT('Report xxxxxxx successfully produced')
ENDROUTINE

Some points to note about this example are:

- It is assumed that the field #REPMODE (report production mode) is defined in the dictionary. There is no need to define it in this program. It probably has a default value of B (batch production).
- The system variables *PROCESS and *FUNCTION used in the SUBMIT commands ensure that this (ie: the current) process/function is invoked by the batch job.
- The system variables *BATCHJOB and *NIGHTJOB are used as the JOBQ (job queue) parameter values in the SUBMIT commands.

It is assumed that these have been previously set up as LANSA system variables prior to coding the program. By using this form to identify the job

queue there is no need to "hard code" the job queue names into the program.

Additionally, the actual values used can be changed externally without having to modify or re-compile the program.

• The built-in function CLR_MESSAGES is used to clear/remove the operating system message which results from executing a SUBMIT command. A more "user friendly" message is issued to replace it.

5.4 Writing More Flexible Reporting Programs Using the OPEN Command

Portability Considerations

The features described in this section are ONLY supported on IBM i.

Report programs can be made more flexible by using the LANSA OPEN command in conjunction with the USE_OPTION(*OPNQRYF) parameter.

The OPNQRYF command is actually an IBM i operating system command. By using it with LANSA you can make reports much more flexible, often replacing several reporting programs with just one program.

The power of the OPNQRYF command is primarily used in two different ways to increase function and flexibility. These are:

- 1. To alter the data selection criteria used by the program at execution time.
- 2. To alter the data ordering criteria used by the program at execution time.

Consider the following example:

REQUEST FIELDS(#REP_TYPE)

```
CASE OF_FIELD(#REP_TYPE)
WHEN VALUE_IS('= A')
CHANGE FIELD(#ORDER_BY) TO('PRODNO CUSTNO PARTNO')
WHEN VALUE_IS('= B')
CHANGE FIELD(#ORDER_BY) TO('CUSTNO PARTNO')
WHEN VALUE_IS('= C')
CHANGE FIELD(#ORDER_BY) TO('PARTNO DEPTNO CUSTNO')
ENDCASE
```

OPEN FILE(PARTHIST) USE_OPTION(*OPNQRYF) KEYFLD(#ORDER_BY) EXECUTE SUBROUTINE(PRINT)

Here the user is asked to nominate what type of report is required (field #REP_TYPE). This value is then used to set up a KEYFLD parameter value in a variable called #ORDER_BY. This value is then passed to the OPNQRYF command via the LANSA OPEN command and the report is produced.

This program can now produce the same report in 3 different orders. By product, customer and part (a type A report), by customer and part (a type B report), or by part, department and customer (a type C report).

The same type of processing can be used to assemble the QRYSLT parameter of the OPEN command to alter the actual data selected by the program from the file for inclusion on the report.

Refer to the *LANSA Technical Reference Guide* for more details and examples before attempting to use the **OPEN** command in this way.

5.5 Overriding Printer File Attributes using a Report Controller

PortabilityThe features described in this section are ONLY
supported on the IBM i.

You can override a printer file's attributes, but not by using the operating system OVRPRTF command directly from EXEC_OS400 or EXEC_CPF commands.

Use a CALL to QCMDEXC passing a command string containing the OVRPRTF command that you wish to use.

For example, write a program like this:

DEFINE #CMD TYPE(*CHAR) LENGTH(100) DEFINE #CMDLEN TYPE(*DEC) LENGTH(15) DECIMALS(5) DEFAULT

CHANGE #CMD TO("OVRPRTF QSYSPRT COPIES(8)") CALL PGM(QCMDEXC) PARM(#CMD #CMDLEN) NUM_LEN(*DEFIN CALL PROCESS(SALES) FUNCTION(REPT01)

Where process/function SALES/REPT01 is the actual function that produces the report.

Note that the override affects file QSYSPRT.

If SALES/REPT01 used a different printer file in a DEF_REPORT command, use this name instead in the OVRPRTF command.

You will also note that the command string #CMD is a variable, so by using SUBSTRING, CONCAT, etc, you can build up command strings that contain varying information (e.g.: the number of copies in the previous example could be sub-stringed into #CMD from a variable or the FORMTYPE modified).

Additionally, this particular function (ie: the one that does the override) could be written as a general purpose program that is exchanged the output queue name, number of copies, etc and the name of the process/function to call to actually produce the report.

For instance, to invoke report process/function SALES/REPT01 to produce 6 copies onto output queue PRT15, you might code the following:

change #rpt_copies 6
change #rpt_outq 'PRT15'
change #rpt_proc 'SALES'
change #rpt_func 'REPT01'

```
exchange (#rpt_copies #rpt_outq #rpt_proc #rpt_func)
then
call process(reports) function(invoke)
or
submit process(reports) function(invoke)
```

where process/function REPORTS/INVOKE is like the first example. It issues the required overrides for copies and output queue and then uses a variable call to invoke the actual reporting program:

```
call process(#rpt_proc) function(#rpt_func)
```

This general purpose routine could be even smarter. Instead of exchanging the output queue name to it, it might look up a table keyed by the user's identification to find his/her's associated output queue.

It could become the report controller for the site or system.

5.6 Portability Considerations

When working with reporting functions with Visual LANSA, you should be aware of the following:

- Applications that "fiddle" with the "page number" field(s) REPnPAGE during the production of a report may not function exactly as they do under IBM i . Under Visual LANSA the page number is set from an internal value and not actually taken at its face value from REPnPAGE.
- The use of the option *FILE for report width, length, overflow, etc has no meaning in a Visual LANSA context. Where *FILE is specified, the defaults from the system definition will be used to calculate form width, length, etc.

6. Pop-up Windows

- 6.1 Locating & Sizing Pop-up Windows
- 6.2 The + in Pop-up Windows
- 6.3 Cursor Location in Output Only Pop-up Windows
- 6.4 Flickering Windows and Program Efficiency for Pop-Up Windows
- 6.5 Using Multiple Pop-up Windows

6.1 Locating & Sizing Pop-up Windows

One of the most time consuming aspects of setting up a pop-up window is deciding upon the location and size of the window required.

If you follow these simple techniques, the amount of time involved can be significantly reduced.

- Always think of where you want the upper left hand corner of the window to be.
- Code this into the command like this:

POP_UP FIELDS(#A #B #C #D) AT_LOC(5 22) WITH_SIZE(*AUTO) The WITH_SIZE(*AUTO) option tells LANSA to size the window to fill all the rest of the screen to the right of and below the AT_LOC row and column positions specified.

• If you then invoke the screen painter, the result would look something like this - which may be considerably larger than you actually need.



• By using the screen painter, you can reorganize the fields in the window into the form you want like this:

row 5 col 22 ----> POP01 Pop-up demo : : Field A IIIIIIII : : Field B II : : Field C IIIIIIIIIII : : Field C IIIIIIIIIII : : Field D III : : Field D III : : Enter F15=Exit F12=Cancel F14=Msgs :

```
:.....
```

Now before exiting from the screen painter, and aided by the "G" (screen grid) painter action, observe that relative to the upper left hand corner of the window you can reduce the window width by about 6 characters and its length by about 5 lines.

• After exiting from the screen painter, the actual AT_LOC and WITH_SIZE parameters are re-written like this (figures shown are approximate and not actual):

```
AT_LOC(05 22) WITH_SIZE(55 19)
```

so you can quickly change them to be like this:

AT_LOC(05 22) WITH_SIZE(51 14)

• Now, if the screen painter was run again, the result should be a nicely proportioned window with its upper left hand corner at row 5 position 22:

6.2 The + in Pop-up Windows

When you place a browse list in a pop-up window, a "+" sign may appear to indicate that more details follow on the next page. The processing involved is identical to that used for full size IBM i screen formats.

You should be aware of the existence and location of the "+" sign as it is very easy to create browse lists that overlay its attribute bytes or overlay it entirely.

If the bottom right hand corner of a pop-up window that contained a browse list was examined, the "+" sign would be located:

DDDDDDDDDDDD DDDDDDDDDD	DDDDDDDD	:
DDDDDDDDDDDD DDDDDDDDDD	DDDDDDD	:
DDDDDDDDDDDD DDDDDDDDDD	DDDDDDDD	:
DDDDDDDDDDDD DDDDDDDDDD	DDDDDDDD	:
DDDDDDDDDDDD DDDDDDDDDD	DDDDDDDD	+:
<messages> :</messages>		
<function 1="" key="" line=""> :</function>		
<function 2="" key="" line=""> :</function>		
:		

If you move the last field in the list one space to the right, its attribute byte will overlay the attribute byte of the "+" sign. This is okay, and everything will work okay, but the "+" sign will lose its high intensity attribute and its color.

If you move the last field in the list two spaces to the right (which is the limit to which you can move it), its attribute byte will overlay the actual "+" sign. This is okay, and everything will still work okay, but the "+" sign will not appear in the window.

6.3 Cursor Location in Output Only Pop-up Windows

When you write your first pop-up window program that is "output only" (ie: no input capable fields) you will probably get a result like this on your first attempt:



This is clearly unacceptable, because the user has to move the screen cursor right across the screen and into the "Companies" pop-up window to select the desired company.

The simplest solution to this problem is: code the POP_UP command like this:

POP_UPwhatever.... CURSOR_LOC(*ATFIELD #COMPNO) which will cause the pop-up window to appear like this with the screen cursor under 0 in "01 ACME ENGINEERING":

DEMO01 Demonstration Panel

Specify the following and press e: COMP01 Companies : : : : Account number . . . : Number Description : : 01 ACME ENGINEERING : : 01 ACME ENGINEERING : : 02 CONSOLIDATED ENG : : 03 ACME WELDING : : 04 DESIGN INC. : : : : Transaction amount . . : : : : : :

6.4 Flickering Windows and Program Efficiency for Pop-Up Windows

By their nature, pop-up windows cause more screen writes to be issued than normal full screen panels resulting from DISPLAY or REQUEST commands. This is particularly true of pop-up windows containing browse lists.

To ensure the maximum efficiency of a program that uses POP_UP commands, and to stop any screen "flickering", force LANSA to use DFRWRT(*YES) in the associated IBM i display file by using either of the following declarative commands in the RDML program:

FUNCTION OPTIONS(*DEFERWRITE) OPEN FILE(*ALL) USE_OPTION(*ONDEMAND)

The significance of the DFRWRT(*YES) option on display files is discussed more fully in the section of this guide on performance considerations.
6.5 Using Multiple Pop-up Windows

Workstations connected to an IBM i use an "attribute byte" on the screen to control a field's attributes. Things controlled by this byte include reverse video, underlining, high intensity, etc.

If you write applications that make frequent use of pop-up windows, you should be aware of the concept of attribute bytes and the cosmetic problems that may result if you overlay one with a pop-up window.

For example, if the following window was displayed on a workstation screen panel:



And then you overlay another window like this:

:....:

Then the actual resulting display will look like this:



The top border of window 1 has disappeared from the display, because window 2 erased its attribute byte from the screen panel.

While this does not cause any program or screen processing problems, it is usually considered to be aesthetically unacceptable.

However, this problem can be prevented if some simple site techniques for window handling are developed and then followed.

Review these three site techniques

Technique 1

Technique 2

Technique 3

Technique 1

Make sure that the windows do not overlap one another. They are either completely separate from one another, or completely replace one another.

While this technique is very effective, it is very boring and largely nullifies the aesthetic appeal of overlapping windows.

Technique 2

Another technique is to always use windows that only overlap each other on the left hand side. Additionally, no window overlaps the top or bottom border of a window on its right.

For example, a program might display windows 1, 2 and then 3. The final display might look like this:



Note however, that any one of the 3 windows can be redisplayed without causing the top or bottom border or any existing window to disappear. For example, redisplaying window 2 would cause a display like this:





If window 1 was then redisplayed, the resulting display would look like this:



Technique 3

This technique merely involves combining techniques 1 and 2 together with a "plan" of the windows and testing each window against some simple rules.

Consider the following example for 4 windows. Any of the windows can be redisplayed at any time without upsetting any other window.

Note how every window, but one, conforms to the following rules.

- Is completely separate from (and therefore does not overlap) another window (e.g.: 4 is separate from 2).
- Or, completely contains another window (e.g.: 1 completely contains 4).
- Or, is completely contained within another windows (e.g.: 4 is completely contained within 1).
- Or, where an overlap occurs, no overlap of the top or bottom LEFT HAND CORNERS of the other window occurs. (e.g.: 4 overlaps 3, but does not overlap its top or bottom left hand corners).



Note that window 1 does not conform to these rules. It overlaps the top and bottom left hand corners of window 3, and therefore could cause the top and bottom border areas of window 3 to "disappear".

7. Prompt Key Processing

- 7.1 What is PROMPT?
- 7.2 What Can the Prompt Key Offer to a System Designer
- 7.3 The PROMPT_KEY Parameter
- 7.4 Prompt Key Processing Programs
- 7.5 What Happens When the PROMPT Key is Used
- 7.6 The Simplest Type of Prompt Key Processing
- 7.7 Even More Complex Prompt Key Processing

7.1 What is PROMPT?

Under the IBM SAA/CUA standards, PROMPT is defined as:

"Prompt is a common dialogue action that assists users to complete entry fields. It can save time for users and reduce the chance of typing errors.

To use prompt, users place the selection cursor on the entry field for which they want a list of possible entries. When they request prompt, a pop-up window appears containing a menu panel with single- or multiple-choice selection fields.

When users select one or more choices from the selection field, the pop-up window disappears. The choice text is placed into the entry field as though users had typed it there."

A prompt dialogue can be better illustrated in more visual terms.

• User is working from an entry panel. They tab the cursor into the company number field and use the prompt function key:

DEMO01	Demonstration Panel
Specify the fol	lowing and press enter.
Account numb	er
Account type .	·····
Company num	ber
Transaction an	10unt

• A pop-up window of all allowable company numbers is displayed. The user selects the desired (company 03) by placing the selection cursor on it and pressing enter:

DEMO01 Demonstration Panel

• The pop-up disappears and the selected company number is placed into the company number field just as if the user had keyed the value in:

DEMO01	Demonstration Panel			
Specify the fol	lowing and press enter.			
Account number				
Account type .	·····			
Company number 03				
Transaction an	nount			

7.2 What Can the Prompt Key Offer to a System Designer

The prompt key processing facilities provided by LANSA offer the system designer the following features: ?

- Vastly improved productivity. No RDML code is required to implement prompt key processing. The link between the field being prompted and the prompting program (that presents the pop up window) is defined in the data dictionary.
- A consistent and easy to use prompting interface.
- Complete compliance with the SAA/CUA guidelines.

7.3 The PROMPT_KEY Parameter

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 area refer to The System Definition Data Areas in the LANSA for iSeries 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 478 479 of the system definition data area. Refer to The System Definition Data Areas in the *LANSA for iSeries User Guide* for full details of the system definition data area.
- In SAA/CUA applications it is assigned from the partition level value assigned to the prompt function key. Refer to Creating or Changing System Partition Definitions in the *LANSA for iSeries User Guide* for more details of how and when this value is assigned.

7.4 Prompt Key Processing Programs

Before attempting to enable and use the PROMPT key on any DISPLAY, REQUEST or POP_UP command you must understand what a prompt key processing program is and how it is used.

The following points elaborate on some of the most important points about prompt key processing programs:

- Prompt key processing programs are just normal RDML functions that have been coded especially to handle prompt key requests.
- Prompt key processing programs normally display a list of allowable values and allow the user to select the one they wish to use. Typically the list is of "codes" such as company numbers, product numbers, customer numbers, department numbers, etc.

Usually additional information is displayed in the list to allow the user to decide which code they wish to use (e.g.: company name, product description, customer name & address, etc).

- Prompt key programs are easily thought of as "black boxes". For instance a prompt key processing program that handles company numbers can be thought of as a "black box". You invoke it, and when it completes execution it gives back a company number (via the exchange list). How it decided upon (ie: "prompted") the company number is immaterial, what is important is that it returns a company number.
- The handling of the prompt key is performed automatically by LANSA in most situations. In our previous example about company numbers, the link between the field "company number" and its associated "prompt key processing program" is defined in the data dictionary. Once this link is defined, LANSA can handle the prompting of field "company number" automatically from any number of different screen panels.

7.5 What Happens When the PROMPT Key is Used

Consider the following input screen generated by an RDML REQUEST command. It requests that the user input a company number, a department number and an invoice number. Note also that the prompt key is enabled (and is handled automatically):

INVOICE01	Invoice Inquiry
Company numbe Dept number Invoice number	er
F4=Prompt	

If the user tabs into the field department number, and uses F4, the following processing occurs:

- The **cursor location** is examined and determined to be within the boundaries of field department number. If the cursor is not within the boundaries of a field, the screen is re-displayed with an error message indicating that the cursor must be within a field's boundaries when the prompt key is used.
- The **definition of field department number** is located in the data dictionary and the associated prompting process/function extracted. If no prompting process/function exists for the field the screen is re-displayed with an error message indicating that the prompt key is not available for the field.
- The following information is posted to the exchange list:
- Field values defined in EXCHANGE OPTION(*ALWAYS) commands
- The current value of department number (i.e., the prompted field)
- If the field department number refers to another field for its definition, then its value is posted again, but under the "referred to" field's 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. These fields should be defined in your data dictionary, allowing you direct reference to them. If not, define them both as alpha (A) fields of length 10. They are supplied to the prompting process/function to allow it to determine which field is being prompted.

- If position 499 in the system definition data area DC@A01 is set to "Y" (EXCHANGE all fields on a prompt request), as many of all the other fields used by the program as will fit into the space left in the exchange list will be posted to the exchange list.
- The **prompting process/function is invoked**. It is a simple program and just displays a list of all departments in a pop up window and allows the user to indicate the desired one by selecting it with the cursor. The screen panel that the user is seeing at this point might look like this:

INVOICE01	Invoice Inquiry
Company number . Dep t number Invoice number :	: DEP01 Departments : : DEP01 Departments : Dept Department : Num : Description :
	001 ADMINISTRATION : 002 FINANCE : 004 SALES & MARKETING : 006 PLANT MAINTENANCE + : :
F4=Prompt :	

The user selects department number 004 (SALES & MARKETING). The prompting function has an EXCHANGE FIELDS(#DEPTNO)
 OPTION(*ALWAYS) command, so the selected value of department number is posted back onto the exchange list. Remember that when the prompting function was invoked it "extracted" all the things it wanted from the exchange list and then cleared it. At this point then, department number is

the only thing on the exchange list.

- **Control returns to the original program**. Since field department number is input capable, the exchange list is examined and the values of all fields known to the original program are mapped back into its storage. In this case, only the department number is mapped back.
- Finally the exchange list is cleared and the screen is redisplayed. Since the value of field department number has been changed by the value returned in the exchange list, the resulting display would now look like this:

INVOICE01	Invoice Inquiry
Company numb Dept number Invoice number	er 004
F4=Prompt	

The "prompt key cycle" has now completed.

Some further things worth noting about key processing techniques and this example are as follows:

- No RDML level coding at all is required in the "invoice inquiry" program to support this processing, it is totally automatic.
- The interaction technique conforms to SAA/CUA standards.
- Field department number can be prompted on any other screen panel, in any other program, in exactly the same way. This produces a very high level of consistency for "end users" of the application system.
- Field department number can even be in a browse list on a screen panel. If prompted, the returned value will be updated into the correct browse list entry before the screen is re-displayed.
- If field department number is on a screen panel, but is not input capable, the prompt key can still be used. However, in this case, any values returned in the exchange list are ignored.

• Consider an inquiry screen that is showing invoice details to the user. All fields on the screen are protected (i.e., not input capable), but the prompt key is still enabled:

INVOICE02 Invoice Inquiry
Company number 003 ACME ENGINEERIN
Dept number 004 SALES & MARKETIN
Invoice number 1627487 Date of issue 10/10/89
Customer 152643 Talls Hardware Shop 121 Any Street Anytown 41828
Product Description SKU Quantity Price 0001920 1/4" BOLTS BOX 300 700.45 0188734 4" CLOUT HEAD NAILS BOX 200 200.34 1102939 ROOFING STRUTS PALLET 2 1107.00 0126378 6" GALVANISED GUTTERIN 20LENG 10 265.45
F4=Prompt

By using the prompt key from this screen, the user might be able to perform the following additional "inquires" (with no extra RDML coding required in the invoice inquiry program) on:

- Companies.
- Departments.
- Other types of invoice inquiries. The same program cannot be invoked recursively by the prompt key. This situation is trapped and the screen

redisplayed with an error message.

- A calendar. Some sites nominate a "calendar" program as the prompt key processing program for all date fields. This allows the user to select the date required from a "calendar".
- Customers.
- Products.
- SKUs. (Stock taking units).

7.6 The Simplest Type of Prompt Key Processing

The simplest types of prompt key processing programs are the ones that display a list of values to the user.

The previous example that handled department number prompts would probably look like this (it does not use page at a time techniques because there are only 20 or so departments):

<< define browse list and exchange list >>

DEF_LIST NAME(#DEPTS) FIELDS(#DEPTNO #DEPTDESC) SEL_ENTRY(#CURSORSEL) EXCHANGE FIELDS(#DEPTS) OPTION(*ALWAYS)

<< clear list and place details of all departments into it >>

CLR_LIST NAMED(#DEPTS) SELECT FIELDS(#DEPTS) FROM_FILE(DEPTFILE) ADD_ENTRY TO_LIST(#DEPTS) ENDSELECT

<< Display list in pop-up window until user selects or cancels >>

CHANGE FIELDS(#DEPTS) TO(*NULL) DOUNTIL COND('#CURSORSEL *GT 0') POP_UP BROWSELIST(#DEPTS) CURSOR_LOC(*ATFIELD #DEP' EXIT_KEY(*NO) MENU_KEY(*YES *RETURN) IF COND('#CURSORSEL *GT 0') GET_ENTRY NUMBER(#CURSORSEL) FROM_LIST(#DEPTS) ENDIF ENDUNTIL RETURN

7.6.1 More Complex Prompt Key Processing

Since prompt key processing programs are user coded RDML programs, they can be very flexible and handle just about any situation.

An example of a more complex prompt key, processing program might involve prompting something like a "customer number".

When invoked, it might initially present a pop-up window that looks like this:

Invoice Inquiry INVOICE01 Company number ... Dept number Invoice number . . . Customer number : CUST01 Customer Number Search : Select type of search required and press enter : : : 1. By name : : 2. By outstanding orders : : 3. By city of residence : 4. By post / zip code : : F12=Cancel F14=Msgs : •



The actual processing that this program performs is immaterial. What is important is that you understand the following points about using it:

- That it can be viewed like a "black box". It is invoked and it returns a customer number. How it does this does not matter very much to the routine which invoked it.
- That it produces a very consistent and reliable user interface. Any screen

panel that requests that a user input a customer number can have the search routine automatically invoked without coding one extra line of code. You would have to take positive steps to be inconsistent.

- That routines that display a customer number (i.e., it is not input capable on the screen panel) still have the ability to invoke the prompt routine automatically. Information exchanged back by the prompting program is ignored in this case.
- That it does not have to use pop-up windows. Normal DISPLAY and REQUEST commands can be used, and may be more suitable, in some cases. The main reason for using pop up windows is for SAA/CUA compliance.

7.7 Even More Complex Prompt Key Processing

Since the LANSA pop up window facility supports input capable windows, it is possible to create prompt key processing programs that do file maintenance via windows.

Consider the following prompt program for department number. Note that it allows the user to select a department, change an existing department, delete an existing department or even create a new department:

INVOICE01	Invoice Inquiry
Company number . Dep t number Invoice number	: DEP01 Departments:: DEP01 Departments:: 1=Select 2=Change 3=Delete :: Dept Department:: Dept Department:: Sel Numb Description:: O01 ADMINISTRATION:: 002 FINANCE:: 004 SALES & MARKET:: 006 PLANT MAINT:: 007 VEHICLE MAINT:: F6=Create F12=Cancel:
F4=Prompt	: : ::

If the user elected to create, change or delete a department, then another pop up window might be used.

The following example might reflect what the screen looks like after using F6 and keying in the new department details:

INVOICE01 Invoice Inquiry



The processing required to do this in the prompting program is all fairly standard. However, once this program is in place, a user can select from or maintain the departments file from any screen panel that has the field department number on it.

By using the CHECK_AUTHORITY built in function, and a little bit more RDML code, the maintenance capability can be excluded from users who are not authorized to use it.

8. Processing Arrays

- 8.1 Introduction
- 8.2 Technique 1 Virtual Fields and a Working List
- 8.3 Technique 2 Using the SUBSTRING Command
- 8.4 Technique 3 Using the DEF_ARRAY Command

8.1 Introduction

The array structure is basically not supported by relational design, and thus always causes some problems when used with any product designed to work with relational database designs.

Most people with a System/38 background have at one time or another had serious problems when attempting to process some form of "array structure" with products like QUERY, PC/TRANSFER or SQL.

In reality however, array constructs are very common in database designs, particularly in applications that have originally come from System/3/34/36 computers.

Additionally, many system designers justifiably include array constructs into design as a legitimate way of improving the performance of an application.

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 database design terms.

The array processing facilities within LANSA are provided as a concession to reality. Array structures within files are very common and provide better system performance than fully normalized relational database designs.

The presence of these facilities 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.

To aid in processing arrays, the following sections describe 3 differing techniques for processing array structures at the RDML level.

Since the newer and more efficient technique 3 was introduced, techniques 1 and 2 are now largely redundant. They remain in this guide for historical documentation only.

For more details about technique 3 you should also refer to the DEF_ARRAY (Define Array) command in the *LANSA Technical Reference Guide*.

In all the following examples, it is assumed that the database file involved contains a character field called CYR that is 84 bytes long.

It actually contains an array of 12 x 7,2 signed positive decimal fields.

Where array constructs contain packed decimal data, slight modifications to technique 2 would be required to ensure that it works correctly.

Techniques 1 and 3 should work on packed decimal data without any major

modifications.

Even though the techniques described only deal with one 12 position numeric array, it should be very easy to extrapolate the techniques to handle more than one array, to handle arrays with a different number of entries, or to handle arrays containing alphanumeric data.

8.2 Technique 1 - Virtual Fields and a Working List

Note: This technique is considered redundant. Refer to 8.4 Technique 3 - Using the DEF_ARRAY Command for the best solution.

Set up virtual fields (CYR01, CYR02, CYR12) and virtual code to return the separate fields from field CYR to the program after reading a record from the file, and to "re-build" the field CYR from the separate fields when writing a record to the file.

Define some virtual code for Data structure specifications ("I" specs).

I DS

Ι	1 84 WRK
Ι	1 72WRK01
Ι	8 142WRK02
"	
"	
Ι	78 842WRK12

Derive some virtual code for Calculations after input from file ("C" specs).

```
C*
C* VC USING FIELDS(CYR01 CYR02 ..... CYR12 CYR)
C*
С
       MOVELCYR
                    WRK
С
       Z-ADDWRK01 CYR01
С
       Z-ADDWRK02 CYR02
С
       .. ..
              "
С
       "
          ...
              ...
С
       Z-ADDWRK12 CYR12
C*
```

Derive some virtual code for Calculations before output to file ("C" specs).

C* C* VC_USING FIELDS(CYR01 CYR02 CYR12 CYR) C* С Z-ADDCYR01 WRK01 С Z-ADDCYR02 WRK02 С 11 Z-ADD " С Z-ADDCYR12 WRK12 С MOVELWRK CYR C*

After reading a record from the file, an RDML subroutine can be executed to place the separate values retrieved in the virtual fields (loaded from field CYR by the virtual code) into a working list, which can then be indexed like an array. This example is working with an "array" of 12 virtual fields called CYR01 -> CYR12.

FETCH FIELDS(#CYR01 #CYR12) FROM_FILE(.....)
EXECUTE LOAD
SUBROUTINE LOAD
DEF_LIST NAME(#LIST) FIELDS(#CYRXX) TYPE(*WORKING) NBR_ENTRYS(12)
CLR_LIST #LIST

CHANGE #CYRXX #CYR01 ADD_ENTRY #LIST CHANGE #CYRXX #CYR02 ADD_ENTRY #LIST " " "

CHANGE #CYRXX #CYR12 ADD_ENTRY #LIST ENDROUTINE

Before writing a record to the file, an RDML subroutine can be executed to place the 12 values in the working list back into the 12 individual virtual fields (which will subsequently be loaded back into the field CYR by the virtual code) that will actually be used to update the database.

EXECUTE ULOAD UPDATE FIELDS(#CYR01 #CYR12) IN_FILE(.....)

SUBROUTINE ULOAD GET_ENTRY 1 #LIST CHANGE #CYR01 #CYRXX GET_ENTRY 2 #LIST CHANGE #CYR02 #CYRXX " " " GET_ENTRY 12 #LIST CHANGE #CYR12 #CYRXX ENDROUTINE

If the file contains more than 1 array, the LOAD and ULOAD routines can be

extended to include more fields in working list #LIST. Thus only 1 working list and 1 pair of subroutines are required.

Once these routines are used to place the 12 fields into a working list you can manipulate the values just like an array.

For example, to calculate total of all 12 fields:

CHANGE #TOT 0 BEGIN_LOOP FROM(1) TO(12) USING(#I) GET_ENTRY #I #LIST CHANGE #TOT (#TOT + #CYRXX) END_LOOP

to increment all 12 values in list by 10 percent:

BEGIN_LOOP FROM(1) TO(12) USING(#I) GET_ENTRY #I #LIST CHANGE #CYRXX (#CYRXX * 1.1) UPD_ENTRY #LIST END_LOOP

8.3 Technique 2 - Using the SUBSTRING Command

Note: This technique is considered redundant. Refer to 8.4 Technique 3 - Using the DEF_ARRAY Command for the best solution.

Note that this method will not work for arrays that contain negative values. Refer to the **SUBSTRING** command in the *Technical Reference Guide*.

In the RDML program use a "get" subroutine to retrieve a value imbedded within field CYR.

SUBROUTINE NAME(GET_FLD) PARMS((#INP *RECEIVED)(#VALUE *RETURNED)) DEFINE #INP *DEC 3 0 (always use odd length packed) DEFINE #VALUE *DEC 7 2 DEFINE #POS *DEC 7 0 CHANGE FIELD(#POS) TO('(#INP * 7) - 6') SUBSTRING FIELD(#CYR #POS 7) INTO(#VALUE 1 7) ENDROUTINE

In the RDML program use a "put" subroutine to place a value back into field FLD.

SUBROUTINE NAME(PUT_FLD) PARMS((#INP *RECEIVED)(#VALUE *RECEIVED)) CHANGE FIELD(#POS) TO('(#INP * 7) - 6') SUBSTRING FIELD(#VALUE 1 7) INTO(#CYR #POS 7) ENDROUTINE

By using these routines you can "index" the field CYR like this, to calculate totals of all 12 fields:

CHANGE FIELD(#TOT) TO(0) BEGIN_LOOP FROM(1) TO(12) USING(#I) EXECUTE GET_FLD (#I #CYRXX) CHANGE FIELD(#TOT) TO('#TOT + #CYRXX') END_LOOP

to increment all values in list by 10 percent:

BEGIN_LOOP FROM(1) TO(12) USING(#J)EXECUTE GET_FLD (#J #INCR)CHANGE FIELD(#NVAL) TO('#INCR * 1.1')EXECUTE PUT_FLD (#J #NVAL)

END_LOOP

8.4 Technique 3 - Using the DEF_ARRAY Command

Note: This technique only uses a small part of the command DEF_ARRAY's full features. Refer to the DEF_ARRAY command in the *LANSA Technical Reference Guide* for more details and examples.

In the RDML program define the array index field (#XX) and then use the DEF_ARRAY command to overlay field CYR with array #CYA:

DEFINE FIELD(#XX) TYPE(*DEC) LENGTH(7) DECIMALS(0)

DEF_ARRAY NAME(#CYA) INDEXES(#XX) OVERLAYING(#CYR TYPE(*SIGNED) TOT_ENTRY(12) ENTRY_LEN(7) ENTRY_DEC(2)

After defining array #CYA you can "index" field CYR like this, to calculate totals of all 12 fields:

CHANGE FIELD(#TOT) TO(0) BEGIN_LOOP FROM(1) TO(12) USING(#XX) CHANGE FIELD(#TOT) TO('#TOT + #CYA#XX') END_LOOP

to increment all values in the array by 10 percent:

BEGIN_LOOP FROM(1) TO(12) USING(#XX) CHANGE FIELD(#CYA#XX) TO('#CYA#XX * 1.1') END_LOOP

9. Action Bars and Pull Down Menus

9.1 What Is an Action Bar & Pull	9.11 Action Bar Accelerator Keys
Down Menu?	9.12 Action Bar Help Processing
9.2 Overview of Action Bar	9.13 Action Bar Security Considerations
Layout And Processing Logic	9.14 Action Bar Application Design
9.3 Overview of Pull Down	Considerations
0.4 CUA 1090 Considerations	9.15 Architecture of Action Bars
and Commitments	9.16 Action Bar Panel Design
9.5 The LANSA Implementation	Considerations
of Action Bars	9.17 Action Bar Programming Considerations
9.6 The Action Bar Control Table	9 18 Action Bar Programming Examples
9.7 Invoking an Action Bar as a	with Frequently Asked Questions
Process	9.19 Action Bars: Another Book You Must
9.8 Using an Action Bar From a Function	Read
9.9 Exit and Cancel Key	
Handling in Action Bars	
9.10 Switching to/from Action	
Bar Logic	

9.1 What Is an Action Bar & Pull Down Menu?

The action bar is the element at the top of the screen that consists of a list of choices representing a group of related actions that the users can request. A group of actions appear in a pull down list when users select an item on the action bar. Action bars control the user's dialogue with the application. ?

The action bar implementation within LANSA is at the process level.

A process, which is typically used as a menu in an SAA/CUA partition, can be made to act as an action bar instead.

This facility is controlled by the "Process Menu/Style" field that is specified when "Creating a new process".

Additionally it can be changed via the "Miscellaneous Process Details" facility. A single line action bar process might appear like this:

File	Edit	Options	Help	
+				

If a user selected the first action bar choice (File), then a pull down might appear like this:

+ File	Edit	Options	Help		+
1. Oper 2. Clos 3. DTX 4. FAX	n e to H/O to H/O]			+

An action bar can use from 2 to at most 4 lines across the top of the panel. Each line can contain at most 6 action bar choices, thus totaling 18 choices within one process.

Each action bar choice has an associated pull down.

Each pull down can support at most 9 choices.

Thus $18 \ge 9 = 162$ choices can be controlled from one action bar.

When a process is used as a normal menu, no extra details are required.

However, when a process is used as an "action bar", additional details about the layout and options in the action bar and its associated pull downs must be specified in a table called the "Action Bar Control Table".

9.2 Overview of Action Bar Layout And Processing Logic

When an action bar process is invoked, it will appear on the top lines of the screen panel like this example:

Note the separator line on line 2. The separator line always appears immediately below the last action bar line. It cannot be suppressed or changed in any way.

Under CUA rules, the last choice in an action bar must be "Help".

Normally the HELP option is handled automatically, but you should be aware of its existence, because it will use up one of the choices in your action bar.

Note how the underlying "panel body" has been pushed down so that it starts on line 3 of the display.

Also note the panel identifier on the left of the 3rd line.

When you are making use of an action bar you will probably find that the panel identifier detracts from the panel's appearance.

To suppress panel identifiers, either suppress all panel identifiers at the partition definition level, or individually, by using the PANEL_ID(*NONE) option on the DISPLAY or REQUEST command that causes the panel body to appear under the action bar.

An action bar can have from 1 to 18 choices. So it can actually push the underlying "panel body" down like this (using 7 to 12 choices):

Action 1 Action 2 Action 3 Action 4 Action 5 Action 6
 Action 7 Action 8 Action 9 Action 10 Action 11 Action 12

Or even like this (using 13 to 18 choices):

```
1 Action 1 Action 2 Action 3 Action 4 Action 5 Action 6
2 Action 7 Action 8 Action 9 Action 10 Action 11 Action 12
3 Action 13 Action 14 Action 15 Action 16 Action 17 Action 18
4 ______
```

5 << Panel Title >>

Options are selected from the action bar by positioning the screen cursor on, or immediately before, the choice and pressing the enter key.

You can move around the action bar by using the cursor movement keys, but generally the quickest way on NPTs (non-programmable terminals) is to use the Tab keys.

Use the Tab keys to tab directly from one action bar choice to another (in either direction).

The tab keys will also allow you to tab back into the "panel body" when only the action bar is on display.

However, once a pull down menu appears on the display, you can only tab around the action bar and pull down menu.

If you tab back into the panel body, you effectively (re)activate the underlying panel (and its function key processing logic).

Other function keys available when working from an action bar would normally include Exit, Cancel, Switch and any enabled Accelerator keys.

These are described in detail in the following sections.

The text that you use to identify each action bar choice should be set up according to the guidelines prescribed by CUA 1989.
9.3 Overview of Pull Down Layout And Processing Logic

When an action bar choice is selected by positioning the screen cursor and pressing enter, a pull down menu of the choices associated with the selected action bar option will appear like this example:

Note the selection field in the upper left corner of the pull down.

It is aligned under "Action 2", which indicates that "Action 2" was selected from the action bar.

This pull down shows the choices that are associated with action bar option "Action 2".

In this example, there are 5 pull down choices.

There may be from 1 to 9 choices in any pull down window. It is a CUA 1989 requirement that each pull down has at least one choice.

The length of the pulldown and therefore the amount of the underlying panel body that it overlays varies with the number of choices that appear in it.

In the leftmost pulldown (or rightmost in a bi-directional language), the last pull down choice should be "Exit".

This will be included automatically by LANSA provided that you have left at least one free space in the first pull down.

Pull down choices are numbered consecutively from top to bottom.

The number appears to the left of the choice (or to the right in bi-directional languages).

If the number is not shown, and is replaced by an "*", it means that the choice is currently "unavailable". You cannot select an unavailable choice or use any accelerator key associated with it. Normally a pull down choice is tagged as unavailable because some other choice must be used first. For instance a pull down choice "Edit a Document" may remain unavailable until you use the choice "Open a Document Folder".

Note that there is a difference between an "unavailable" choice and one that you are not authorized to use.

When you are not authorized to use a pull down choice, it disappears altogether from the pull down menu, and any following choices bunch up and re-number to cover the gap it leaves.

Refer to 9.13 Action Bar Security Considerations for more details.

RDML functions can control the availability and unavailability of pull down choices by using the SET_ACTION_BAR built in function.

At the top left (or right for bi-directional languages) a selection field appears.

This can be used to enter the number associated with the pull down choice that you wish to use.

Pull down choices can also be selected by positioning the cursor on the same line as the required choice and pressing enter. The cursor must remain within the confines of the pull down menu's borders if this option is used.

The third choice, shown in the example pull down, has the text "F16" on the right of the choice text.

By the CUA 1989 standards, this text indicates that function key 16 can also be used to select this pull down choice.

Function key 16 is called an "accelerator key".

Not only can you use it to select the choice when the pull down menu is displayed, but also directly from the action bar (without bothering to show the pull down), and also directly from any underlying panel/function, without even bothering to switch to the action bar.

Accelerator keys can be used by experienced users to "fast path" through applications without having to bother with the action bar or pull down choices and interactions.

Where an accelerator key is associated with a pull down choice you must include the CUA 1989 prescribed "Fnn" text manually. This feature is not automatically implemented.

You can move around the pull down and action bar by using the cursor movement keys, but generally the quickest way on NPTs (non-programmable terminals) is to use the Tab keys. Use the Tab keys to tab directly from the pull down back to, and around, the action bar in either direction.

The tab keys will not allow you to tab back into the "panel body" when the pull down is on display.

If you tab back to the action bar and press enter you will cause another pulldown to be selected. The current pull down will disappear and be replaced by the new one you selected.

Even if you force the cursor back into the panel body, you still cannot (re)activate the underlying panel. You must cancel the pull down or switch back to the panel body first.

Other function keys available when working from an action bar would normally include Exit, Cancel, Switch and any enabled Accelerator keys.

These are described in detail in the following sections.

The text that you use to identify each pull down choice should be set up according to the guidelines prescribed by CUA 1989.

9.4 CUA 1989 Considerations and Commitments

The LANSA implementation of action bars conforms to, and is committed to, the action bar techniques prescribed by the CUA 1989 standard for NPTs (non-programmable terminals).

In CUA terminology, the implementation conforms to the "Text Subset" of the CUA 1989 "Graphical Model".

It is the highest level of CUA implementation achievable on NPTs.

This means that you should carefully consider whether you are also committed to the CUA 1989 standards for the "look" and "feel" of your action bar applications.

The commitment to the "look" is easy to make, but the commitment to the "feel" may involve changes to the way applications are designed and implemented.

These changes will effect your development staff and your end users alike. Both groups may require education about the new standards being used.

Without this commitment, attempts to design and implement an action bar driven system will almost certainly fail to achieve the best possible result.

The key component of the CUA 1989 standards for action bar driven systems is that of the "Object-Action" approach to application design.

Please refer to the IBM supplied CUA manual *Common User Access:* LANSA Basic Interface Design Guide (SC26-4583) for more details of the Object-Action approach to design. Refer especially to the chapter *Action Bar and Pull-Downs* and the Appendix *Designing an Object-Action Orientated Application*.

Unless you are committed to CUA 1989, and are prepared to possibly alter the way you design midrange application systems, you will find that you are generating large amounts of RDML code to "fight against" the natural flow and architecture built into the LANSA implementation of action bars (in support of CUA 1989).

Finally, please note that you do not have to use action bars.

Sometimes the presence of new facilities and techniques makes people feel bound to use them otherwise they are being "old fashioned".

This is not true of action bars.

Some applications may not even be suitable for action bar implementations.

Traditional menu driven applications that conform to the CUA 1989 "Entry Level" model are perfectly valid, and often the best choice, for some types of applications.

9.5 The LANSA Implementation of Action Bars

LANSA implements action bars at the process level.

This means that a process, which normally appears as a menu, can actually be presented to the user as an action bar instead.

Hopefully, you are familiar with the idea of a process being presented as a menu of "functions" (ie: RDML programs). From the list of functions shown on the process menu, you select the one you wish to use and it is then executed.

A process menu with 3 functions could be visualized like this:

TESTPROC Test Process

Enter number of function or place cursor on line, press Enter.

1. Function 1
2. Function 2
3. Function 3

By defining this process with "Process Menu/Style" as ACT/BAR (rather than SAA/CUA), it could be arranged into an action bar like this:

Action 1

| 1. Function 1 | | 2. Function 2 | | 3. Function 3 |

or like this (with the pulldowns shown for each of the 3 action bar choices):

```
Action 1 Action 2 Action 3
```

| 1. Function 1 | 2. Function 3 | | 1. Function 3 |

Action 1 Action 2 Action 3

| | 1. Function 3 | | 2. Function 1 | | 3. Function 2 |

Setting up a process to work as a menu is easy.

Setting up a process to work as an action bar is more complex because the action bar options, pull down choices and the actual RDML functions that are invoked when they are used must be defined to LANSA.

The information that defines the content and layout of an action bar is contained in a table called the "Action Bar Control Table".

This is described in detail in 9.6 The Action Bar Control Table.

9.6 The Action Bar Control Table

The information that defines the content and layout of an action bar is contained in a table called the "Action Bar Control Table".

Each process of type "ACT/BAR" has its own entries in the action bar control table.

The content of the table can be reviewed and changed at any time by using the option accessible from the Process Definition Menu.

If you attempt to review the action bar control table entries associated with a process that is not of type "ACT/BAR", a message will display indicating that the process does not have action bar control table entries.

In this situation you can use the "Miscellaneous Process Details" facility accessible from the Process Definition Menu to change the process type from SAA/CUA to ACT/BAR.

Take care if making such a change, because all the associated RDML functions will need to be recompiled to allow them to display the action bar on lines 2 through to at most 4 of the screen panel.

The action bar control table defines the content and layout of the action bar and its associated pull downs as follows:

Action Bar Option Definition

Up to 18 action bar options can be defined.

Each action bar option needs the following attributes defined to make it usable:

A/Bar Keyword (Text Identifier)

Specifies the text that is to appear in the action bar to identify this action bar choice. Some things to note are:

- Try to use just one word.
- Use upper and lower case characters. -
- Support for bi-directional and DBCS languages is provided.
- Conform to the CUA 1989 guidelines.
- CUA has guidelines for standard options such as "File" and "Edit".
- The help option is automatic. You do not have to define it.

Sequence

Specifies the sequence that the action bar option is to appear in the action bar. The number specified must be in the range 1 to 18 and unique within the action bar.

AB\$OPT Field Value

AB\$OPT is a field defined in the LANSA data dictionary. Some things to note about the value you specify here are:

- If it isn't in the dictionary, define it as alpha (length 3).
- The value you specify here is placed into field AB\$OPT when this action bar choice is used.
- Do not use values CUR or ALL, as they are reserved to mean "current" and "all" to built in function SET_ACTION_BAR.
- The field is accessible to RDML functions.
- It allows them to decide exactly which action bar choice was used to cause them to be invoked.
- One RDML function handling multiple action bar choices can have good performance implications. Refer to 9.18.4 Sample Program: All 3 Functions in One Program for an example of this programming technique.
- Value specified should be unique within this action bar.
- Standards for AB\$OPT values should be established.

9.6.1 Pull Down Definition (for Each Action Bar Option Defined)

Each action bar option defined **must** have an associated pull down menu.

The pull down menu can have from 1 to 9 choices defined in it.

Each individual pull down choice must have the following things defined to make it a usable choice:

No (Sequence Number)

Specifies the number associated with the pull down choice. It cannot be changed. To reorder choices in a pull down you must over type the details. Deleted choices cause following choices to move up so that all choices are always numbered consecutively.

Description

Specifies the text that is to appear in the pull down to identify this pull down choice. Some things to note are:

- Use upper and lower case characters.
- Support for bi-directional and DBCS languages is provided.
- Conform to the CUA 1989 guidelines.
- CUA has guidelines for standard choices.
- Include "Fnn" to identify accelerator keys (where required).
- Include "..." ellipses for resulting pop-ups (where required).
- The help pull down is automatic. You do not have to define it.

AK (Accelerator Key)

Specifies the accelerator key that is to be associated with this pull down choice. Some things to note are:

- Accelerator keys are optional.
- Use values in range 01 to 24 only to identify an accelerator key.
- Avoid conflicts with other key assignments. This is **not** checked.
- Avoid overuse. They will confuse users and complicate the system.
- Conform to the CUA 1989 guidelines.
- They are activated on **any** panel showing this action bar.
- Develop standards for accelerator key assignments and use.

PD\$OPT Field Value

PD\$OPT is a field defined in the LANSA data dictionary. Some things to note about the value you specify here are:

- If it isn't in the dictionary, define it as alpha (length 3).
- The value you specify here is placed into field PD\$OPT when the pull down choice is used.
- Do not use values CUR or ALL, as they are reserved to mean "current" and "all" to built in function SET_ACTION_BAR.
- The field is accessible to RDML functions.
- Allows them to decide exactly which pull down choice was used to cause them to be invoked.
- One RDML function handling multiple pull down choices can have good performance implications. Refer to 9.18.4 Sample Program: All 3 Functions in One Program for an example of this programming technique.
- Value specified should be unique within this pull down, and preferably, within the entire action bar.
- Standards for PD\$OPT values should be established.

IA (Initial Availability)

Specifies whether or not this pull down choice is to be made available on the initial invocation of the action bar. Some things to note about this option are:

- Leave as blank or enter Y to specify initial availability.
- Specify as N to cause initial non-availability.
- Unavailable pull down choices are shown in blue and have their associated selection numbers replaced by an "*".
- Unavailable and "not authorized" are **not** the same thing.
- RDML program access to make pull down choices available/ unavailable is provided by the SET_ACTION_BAR built in function.

Associated With

Specifies which function/process/special entry is to be invoked when the pull down choice is selected from the pull down menu.

The following is worth noting about this link between the action bar control table and the function/process/special entry it causes to be invoked:

• An RDML function from the current process can be invoked from the pull down menu. This is by far the most common type of action that results from using a pull down choice. The function will have an action bar identical to

the action bar that caused it to be invoked.

- An RDML function from another process can be invoked from the pull down menu. This is not a commonly used technique, because the function may not have an action bar, or if it does, it will be the action bar of its parent process, not the action bar that actually invoked it. The non-appearance, or change in layout and content, of the action bar can be confusing to developers and end users alike, explaining why this technique is rarely used.
- Another RDML process can be invoked from the pull down menu. The invoked process may appear as a menu, or as another action bar, depending upon how it was defined. This facility allows multiple action bars to be built in a "hierarchy".
- A special entry can be invoked from the action bar. The special entry facility (described in the *LANSA for iSeries* User Guide) allows any IBM i command to be invoked from an action bar, including calls to 3GL application programs.
- The same function/process/special entry can be used in more than one pull down choice.
- What happens when the selected choice is used depends entirely upon the function/process/special entry.
- Being invoked from an action bar has panel design ramifications for RDML functions. For instance, a function will effectively lose the use of from 2 to 4 lines on the top of the screen panel to allow the action bar to be displayed.

The following table indicates amendment actions that can be made to the action bar control table and the type of (re)compilation activity that would normally be required:

Type Of Change To The Action Bar Control Table	Objects To (Re)Compile
More action bar options cause another line to be used on all screen panel(s).	All associated RDML functions. Process (if compiled).
More action bar options do not cause another line to be used.	Process (if compiled).
More pull down choices.	Process (if compiled).
Action bar descriptions and any other action bar	Process (if compiled).

value.

Pull down descriptions and any other pull down Process (if compiled). value.

9.7 Invoking an Action Bar as a Process

When you elect to use a process that is of type "ACT/BAR", its action bar control table entries are examined and used to assemble an action bar.It will clear the screen, and then appear like this (1 to 6 action bar options):Action 1 Action 2 Action 3 Action 4 Action 5 Action 6

Or, like this (7 to 12 options):

Action 1Action 2Action 3Action 4Action 5Action 6Action 7Action 8Action 9Action 10Action 11Action 12

Or even like this (13 to 18 options):

Action 1 Action 2 Action 3 Action 4 Action 5 Action 6 Action 7 Action 8 Action 9 Action 10 Action 11 Action 12 Action 13 Action 14 Action 15 Action 16 Action 17 Action 18

Some things that should be noted about the invocation of an action bar process are:

- The lower portion of the screen (below the action bar) is erased. When an action bar process is invoked it erases the screen, rather than overlaying what is currently on the screen. This gives the user the visual feedback that they have "started something new" (ie: are working with a new action bar).
- Choices may be unavailable. The availability/unavailability is established from the action bar control table (initially) and then controlled by invoked RDML functions by using the SET_ACTION_BAR built in function.
- The Exit function key will be enabled (unless specifically disabled at the process level as part of the partition definition). If used it will act just as it would if the process was presented as a menu.
- The Cancel function key will be enabled (unless specifically disabled at the process level as part of the partition definition). If used it will act just as it would if the process was presented as a menu.

9.8 Using an Action Bar From a Function

When an RDML function that is part of an action bar process is invoked, any screen panel it presents by DISPLAY or REQUEST commands, will have an action bar on its top lines like this (using from 1 to 6 action bar options):

Action 1 Action 2 Action 3 Action 4 Action 5 Action 6

<< RDML Function Panel Title >>

or, like this (7 to 12 options):

Action 1 Action 2 Action 3 Action 4 Action 5 Action 6 Action 7 Action 8 Action 9 Action 10 Action 11 Action 12

<< RDML Function Panel Title >>

or even like this (13 to 18 options):

Action 1 Action 2 Action 3 Action 4 Action 5 Action 6 Action 7 Action 8 Action 9 Action 10 Action 11 Action 12 Action 13 Action 14 Action 15 Action 16 Action 17 Action 18

<< RDML Function Panel Title >>

Some things that should be noted about executing an RDML DISPLAY or REQUEST command within an action bar process are:

- You cannot stop the action bar from appearing.
- Pull down Choices may be unavailable. The availability or unavailability is established from the action bar control table (initially) and then controlled by invoked RDML functions by using the SET_ACTION_BAR built in function.
- The Exit function key will be enabled from the EXIT_KEY parameter of the DISPLAY or REQUEST command that presented the panel and action bar and handled normally.
- The Cancel function key will be enabled from the MENU_KEY parameter of the DISPLAY or REQUEST command that presented the panel and action

bar and handled normally.

- The switch (to action bar or from action bar) function key is always enabled and cannot be suppressed. If the cursor is above or on the action bar separator line when the switch key is used, it is taken as "switch to panel body", if it is below the separator line, it is taken as "switch to action bar".
- The user cannot be stopped from switching to the action bar and altering the flow of control into another RDML function without the "active" RDML function ever regaining **active** control.
- The previous point brings the first touch of the "non-procedural" component of action bar processing. When an RDML function executes a DISPLAY or REQUEST command, it may never regain control back from the screen panel, because the user has caused it to terminate and then activated another RDML function.
- This is an important point to consider when including calls to other functions. If a screen panel is displayed from a called function the user has the opportunity to return to the action bar and re-select the original function. This would cause a recursive call error.
- This non-procedural flow of control means that a different style of application design and programming techniques may be required. Action bar functions tend to be very small, highly independent and able to work "standalone" (ie: without ever relying on another function).
- Action bar functions often appear to be "endless loops", which just perform one simple action again and again, waiting for the user to alter the flow of control into some other function.
- As a result of the amount of control that the user is given, action bar functions tend to have very little "hard coded" flow of control logic (such as function key handling).

9.9 Exit and Cancel Key Handling in Action Bars

When an action bar **process** is invoked, the exit and cancel keys are handled as follows:

- Unless the Exit facility is globally disabled at the partition level, it will be enabled and treated just as if the exit function was used from a process menu.
- Unless the Cancel facility is globally disabled at the partition level, it will be enabled and when used from the action bar, treated just as if the cancel function was used from a process menu. When used from a pull down, it cancels the pull down only, causing the action bar to remain active on the screen panel.
- Using the automatic Exit choice from the left most pull down is functionality equivalent to using the Exit function key.

When an RDML **function** executes a DISPLAY or REQUEST command that presents a panel body and an action bar, the exit and cancel keys are handled as follows:

- Exit key enablement is determined from the EXIT_KEY parameter. If exit is used from the action bar, it is handled normally by the RDML logic as specified by the EXIT_KEY parameter.
- Cancel key enablement is determined from the MENU_KEY parameter. If Cancel is used from the action bar, it causes the action bar to be canceled and the panel body to be redisplayed.

If cancel is used from a pull down, it cancels the pull down only, causing the action bar to remain active on the panel.

• Using the automatic Exit choice from the left most pull down is functionality equivalent to using the Exit function key.

9.10 Switching to/from Action Bar Logic

When an action bar **process** is invoked the switch function key is disabled because there is no panel body present to switch control to.

When an RDML **function** executes a DISPLAY or REQUEST command that presents a panel body and an action bar, the switch function key is handled as follows:

- The switch key is always enabled and cannot be disabled.
- If the screen cursor is on or above the action bar separator line, the use of the switch key is interpreted as "switch to panel body". The screen cursor is relocated to last known position in panel body.
- If the screen cursor is below the action bar separator line, the use of the switch key is interpreted as "switch to action bar".
- Manually moving the screen cursor from an active panel body up into the action bar area and pressing enter is treated as a "switch to action bar" followed by an "enter" request.

If cancel is used from a pull down, it cancels the pull down only, causing the action bar to remain active on the panel.

If cancel is used from a pull down, it cancels the pull down only, causing the action bar to remain active on the panel.

The disadvantage of this technique over using the "switch" key is that no record of the original (and restorable) cursor location in the panel body is available, should the user choose to switch back to the panel body again.

• Manually moving the cursor from an action bar down into the panel body and pressing enter is treated as a "switch to panel body" request followed by an "enter" request.

This rule does not apply when a pull down is on display, only when just the action bar is displayed. When a pull down is on display you can only switch back to the panel body by using the switch key.

This technique is a very fast way of switching from an action bar back to the panel body and activating it by pressing enter. However, please note that it switches back to the panel body **and** presses enter.

9.11 Action Bar Accelerator Keys

When an action bar **process** is invoked the function keys associated with all available pull down choices are enabled and usable.

Using an accelerator key is functionally equivalent to selecting the action bar option, then selecting the choice from the pull down.

When an RDML **function** executes a DISPLAY or REQUEST command, all accelerator keys associated with available pull down choices are enabled and usable:

• They cannot be selectively enabled or disabled by the RDML function, other than by altering the pull down choice availability by using the SET_ACTION_BAR built in function.

If you are using SET_ACTION_BAR for such a purpose you may find that you have some misunderstandings about the purpose and use of accelerator keys and action bars that are overly complicating your design.

- The use of an accelerator key cannot be intercepted by the RDML function.
- Avoid overuse of accelerator keys. Overuse and/or poor standards for assignment can lead to conflicting useand overly complex systems.

Assign accelerator keys only to very commonly used pull down choices.

• Develop and enforce standards for accelerator key assignments and use.

9.12 Action Bar Help Processing

The following information about the help action bar choice, its pull down options and help key processing within an action bar may assist your understanding of action bar processing:

• The action bar option Help is normally automatically added to the action bar.

This may increase the number of options in your action bar so that it actually requires an extra line on the top of each screen panel.

- The pull down that appears when the automatic action bar help option is chosen is also automatically generated and handled.
- Once enabled, automatic help processing cannot be selectively suppressed.
- The text for the word "Help" in the action bar is derived from the short form description of the Help function key, so it should appear in the correct language.
- The automatic help action bar option is enabled if the current partition contains a process called HELP.* \$\$AB.
- The automatic pull down choices that appear are enabled by the presence of the following functions within process HELP\$\$AB:

Pull Down Choice Function Name

Help for Help	HELP\$HH
Extended Help	HELP\$HE
Keys help	HELP\$HK
Help Index	HELP\$HI
Tutorial	HELP\$HT
About	HELP\$HA

- The order of the options in the pull down is pre-determined by the CUA standards and cannot be changed.
- The text for the pull down options is derived from the various descriptions of the RDML functions HELP\$HH through HELP\$HA. he function descriptions can be modified so that the pull down text is presented in the correct language.

• When an automatic help pull down choice is selected, it causes processing as follows:

Pull Down Option	Description of Processing When Chosen
Help for Help	The help text associated with function HELP\$HH from process HELP \$\$AB is displayed.
Extended Help	The help text associated with the active process or function is displayed.
Keys help	The help text associated with function HELP\$HK from process HELP \$\$AB is displayed.
Help Index	The normal help index associated with the active process or function is displayed.
Tutorial	The help text associated with function HELP\$HT from process HELP \$\$HT is displayed.
About	The help text associated with function HELP\$HA from process HELP \$\$HA is displayed.

- Help text can be made to appear in the desired language(s) using the normal multilingual facilities of LANSA.
- The help text is normally presented in a pop-up window. However, if the option to use IBM i Office help is used, this is what will appear.
- Use of the help function key will cause the extended help text associated with the currently active process or function to be displayed, regardless of whether or not the help pull down is on display.

9.13 Action Bar Security Considerations

When a pull down choice will cause an RDML function to be invoked, it is subject to normal LANSA function level security (if enabled).

When a user is not authorized to use a function, the associated pull down function will not appear in the pull down.

This fact is invisible to the user, because no "gaps" appear in the pull down menu. The unauthorized choice disappears and the following choices bunch up and re-number to cover the gap that it has left.

Please note that an "unauthorized" choice is not the same as an "unavailable" choice.

Unavailable choices are flagged by an "*" and are normally unavailable because some other choice must be used first (e.g.: "Edit Document" will be flagged as unavailable until "Open Folder" is used).

If security checks cause all the pull down choices associated with an action bar option to "disappear", then the action bar option itself will disappear from the action bar.

Again, this fact will be invisible to the user, because the action bar will bunch up, leaving no gaps.

This means that even though two users may be using the same action bar process, they may be seeing very different action bar and pulldown choices because they have different levels of access to the underlying functions being used.

This feature is important in multi-user system designs because it means that one action bar structure can actually be "tailored" for each user by using the normal LANSA security facilities.

9.14 Action Bar Application Design Considerations

It is hard to understand how flexible and functional an action bar application can be by just reading guides and manuals. The only real way to find this out is to set up a small prototype system and try it out. How to do this is described in Prototype an Action Bar.

The following points are things that you should consider when planning or designing applications that are to use action bars.

- Unless you are committed to **CUA 1989**, and are prepared to possibly alter the way you design midrange application systems, you will find that you are generating large amounts of RDML code to "fight against" that natural flow and architecture built into the LANSA implementation of action bars (in support of CUA 1989).
- Not all types of applications are suitable for action bar implementations. Refer to the CUA 1989 guidelines following for more details in assisting you to make this decision.
- Action bars require more frequent, but usually more compact, screen panel interactions. They may not be suitable for use over slow communications lines. However, in some instances, the compact nature of action bars and pull downs may actually be more suitable on a slow communications line than traditional menu driven applications. There is only one way to find out the suitability of an action bar implementation on a communications link: implement a small prototype application and try it out.
- **Never** attempt to design a full scale action bar system at your first attempt. Always attempt a small prototype system first to gain the required design and programming experience and to judge the suitability of the application for an action bar approach.
- •

Prototype an Action Bar

1. Create a process of type ACT/BAR, and include into it one RDML function called PROTO, like this:

DISPLAY FIELDS(#AB\$OPT #PD\$OPT) IDENTIFY(*DESC) PANEL_ID(*NONE)

2. After keying in the RDML command, compile function PROTO to assemble the prototype action bar. This dummy function shows the values of the special AB\$OPT and PD\$OPT fields from the action bar and pull down option

chosen.

- 3. Define in the action bar control table all required action bar options, and all associated pull down choices so that they **all** cause function PROTO to be invoked.
 - Once this has been done the action bar exists and can be used. When any pull down option is chosen it will invoke PROTO showing you the chosen action bar option code and pull down choice code.
 - This allows the complete action bar to be assembled, visualized and checked for completeness **before** generating any full scale RDML functions.
- 4. The prototype can then be gradually converted to a complete application by successively generating the required RDML code for each pull down option, and replacing action bar control table references to PROTO with the "real" function.

Including other advanced CUA features such as prompt key processing into an action bar application design is vital if the design is to be simple, yet highly functional.

9.14.1 Applications Least Suitable for Action Bar Implementation

9.14.2 Applications Most Suitable for Action Bar Implementation

9.14.1 Applications Least Suitable for Action Bar Implementation

From the IBM Manual "Systems Application Architecture: Common User Access Basic Interface Design Guide" (SC26-4583):

" Although the text subset of the graphical model is generally preferred for all new applications on the non-programmable terminal, there are some applications whose structure and objectives may warrant an exception to that rule. They are:

An application with only limited actions that can be performed on the primary object or objects in the application. This is true of many data-entry-intensive applications where the only application action available is 'Enter data'. This type of application does not need an action bar because only a limited number of actions are available to users. Either the actions or the objects in this type of application may be implicit, or the actions and objects are combined. These limited-action applications typically are designed to maximize the resources of a host system by minimizing the required number of host interrupts.

A new version of an existing action-object-oriented application does not involve a major re-design of the application. One of the prime objectives for this style of application might be to retain consistency with previous versions for users who will be upgrading to a new version. However, components being used in the application, such as 'entry fields' and 'selection fields', should be revised to conform to CUA. These applications also might have been designed to minimize the number of host interrupts generated by the application.

An application that should be an action-object, as determined by a thorough task analysis. This decision may have been made because the intended users have a current conceptual model of the application that is action-object.

An application that is designed to be 'walk-up-and-use' by users who are constantly new, and, therefore, will not be trained. Such applications must control the users' choices very closely. An example is an information display in a hotel or airport lobby. These applications typically combine object and action choices. They might be object-action or action-object, but they would not include an action bar and, therefore, would be entry model applications. "

9.14.2 Applications Most Suitable for Action Bar Implementation

From the IBM Manual "Systems Application Architecture: Common User Access Basic Interface Design Guide" (SC26-4583):

" For an application with numerous available actions and/or an application whose users need to understand the objects and their relationships, the objectaction orientation is recommended. Some of the major reasons for this recommendation are:

The object-action orientation avoids what are often referred to as 'action modes', which force users to switch from one mode to another to perform different actions on the same object.

In a large system or an application with many actions, the action-object approach may require clustering of actions into higher levels, or categories, of action types, This often results in very large, complicated menu hierarchies. With the object-action approach, only the actions applicable to the users' chosen objects are presented. This avoids the need for large menu hierarchies.

The object-action orientation combined with the visual approach used in CUA supports user exploration and user control, because only the actions applicable to selected objects are available to users. Users can view all available actions and perform them in the desired order, rather than in an order dictated by the application.

In the action-object orientation, actions that apply to multiple objects may have to be given slightly different names, for example, synonym verbs, to distinguish their uses. Users may be confused by the similar action names or be led to believe that more actions are available for a given object than there actually are. With the object-action orientation, the same action name can be used for all objects to which it applies. The application knows how to tailor the action to the chosen object; users, therefore, do not have to distinguish among synonyms.

The larger the number of possible actions, the more difficult it is to refine and nest action types and names consistently and unambiguously. The general categories into which finer distinctions and actions are grouped may not be very meaningful in themselves. They often exist only for the purpose of grouping and may be arbitrary, abstract, or redundant. The object-action orientation, as noted previously, allows the application to offer only the actions applicable to the users' chosen objects. Hierarchies of action types and actions can therefore be minimized."

9.15 Architecture of Action Bars

Action Bar Methodology is different from normal SAA/CUA menus. They must be approached in a different manner. To create a functional Action bar it is not as simple as just changing the process type to ACT/BAR and filling the Action Bar Control Table. Functions used in an AB process are intrinsically different from those used in an SAA/CUA process. If you want the AB to be more procedural you are doing it wrongly.

The AB controls the calling of functions or other actions that might be defined in the Action Bar Control table. An example of why AB might be used is if a user wanted to perform several actions on an item. An AB would be set up to Create, Print and Order items from a pull down menu on the AB.

An AB results in a pull down when an option is selected. Once the pull down is displayed the user can then start an action by selecting from the pull down window. Either an action or a function can be initiated from the pull down. The invoked function should never be an action bar unless it is a popup. By selecting a popup window function the user is taken to another level. Popups do not require a menu or AB, therefore, the AB is virtually unaffected. Whereas, a CALL to a full screen DISPLAY would change the AB menu and be very confusing to the user. When a popup is invoked from an AB the AB options are displayed but cannot be chosen.

After invoking an action, the action bar of the process to which the function belongs will be displayed. This is true even for attached functions.

Following is a schematic of using Action Bars.

Examples

Example 1: Select Option 1, Action 1 (Func A) from pulldown. Func A CALLS Func C from same process.

Process: AB1 Action Bar One





Example 2: Select Option 1, Action 4 (Attached Process/Function), Changes AB and Options.

Process: AB2 Action Bar Two



9.16 Action Bar Panel Design Considerations

The following points are things that you should consider when planning or designing screen panels to use in conjunction with an action bar:

- The action bar may push the "panel body" that you are working with down to line 3 (1 to 6 action bar choices), 4 (7 to 12 action bar choices) or even 5 (13 to 18) action bar choices.
- The automatic "Help" action bar choice will use up one of your available action bar choices. So if you plan on 6, you may actually get 7, thus removing one more line from the available "panel body" that you have to work with.
- Consider suppressing panel identifiers at the partition level or by individual use of the PANEL_ID(*NONE) parameter on the DISPLAY and REQUEST commands that form the "panel body".
- The text that you use to identify each action bar option and pull down choice should be set up according to the guidelines prescribed by the CUA 1989 standard.
- Where an accelerator key is associated with a pull down choice you must include the CUA 1989 prescribed "Fnn" text manually. This requirement is not automatically implemented.
- Full multilingual support is provided for all end user visible text in action bars, including DBCS and bi-directional languages.
- Action bars and pull down menus used in bi-directional language systems are effectively "reversed".

9.17 Action Bar Programming Considerations

The following points are things that you should consider when planning or designing programs to use in conjunction with an action bar:

- Action bar processing programs are generally different to normal entry model programs. They tend to be very small, have one and only one function, and have very little "hard coded" flow of control logic.
- The overriding rule for action bar programming is that the "user is in control". This will ultimately mean that the programs are a lot easier to develop and maintain, but may cause some initial confusion because of the apparent programmer lack of control over what is happening.
- Do not trust / expect other functions to be executed before or after the one you are working on.

Make it robust and able to work standalone.

- Keep the routines simple and short.
- Expect, and allow for, non-procedural flow of control.
- If you are enabling a lot of function keys on DISPLAY and REQUEST commands via the USER_KEYS parameter and coding IF_KEY commands to handle theiruse, consider redesigning your system to be more object orientated and more in the users hands, rather than the designer's hands, with regard to flow of control.

9.18 Action Bar Programming Examples with Frequently Asked Questions

The following examples are designed to be generic.

They deal with a mythical file called OBJECT, in which each record is uniquely identified by a key called OBJECTKEY, and contains information about the object in fields called OBJECTA01 through to OBJECTAnn.

The identification of the "object" that your action bar application deals with is strictly beyond the scope of this guide.

Refer to the IBM supplied CUA 1989 Basic Interface Design Guide for an introduction to object identification in application design.

Additionally, the following examples make no attempt to cover all possible types of applications that can be enabled in an action bar.

The examples are **simplistic** attempts to explain some basic programming considerations and the "philosophy" that should be behind well designed, simple to develop, simple to maintain and simple to use action bar applications.

9.18.1 Sample Program: Selecting An Object

```
EXCHANGE FIELDS(#OBJECTKEY) OPTION(*ALWAYS)
BEGIN_LOOP
REQUEST FIELDS(#OBJECTKEY) IDENTIFY(*DESC)
CURSOR_LOC(*ATFIELD #OBJECTKEY)
END_LOOP
```

Q & A

What does the program do?

It does one and only one thing. Allow the user to identify the object that they wish to work with by typing in its "key".

What if the user doesn't know the "key" value?

Then they would use the prompt key to search through the list of objects in some order and choose the one they wish to work with. This feature is available via normal LANSA prompt key handling described in other sections of this guide. Inclusion of other advanced CUA features like prompting is vital to simple, yet highly functional action bar applications.

Can I validate the "key" that they specify?

You can, but the validation logic would only be executed if the user pressed the Enter key.

It is highly likely that the user will type in the "key", move the cursor up to the action bar and select the "action" that they wish to take on the object, before pressing enter.

In this case, this particular function never regains control after executing the DISPLAY command. The user chooses another function, causing this one to end, and the new one to start.

Does this mean an invalid "key" can be accepted?

It sure does. However, this is not a problem because all functions in an action bar are designed to be highly robust, self contained applications that never trust the fact that another function has been run before them, or will run after them. So if the user keyed in an invalid "key", moved to the action bar and chose something like "Display Object", then the "Display Object" routine would quite easily handle the "not found" situation with an appropriate error message (e.g.: Object to be displayed cannot be found ... select desired object')

Why is the CURSOR_LOC(*ATFIELD #OBJECTKEY) parameter used?

You will find that you have to use the CURSOR_LOC parameter on most DISPLAY or REQUEST commands that are part of an action bar process.

The reason is simple, if you leave the parameter off the command, the operating system will position the cursor up to the first option in the action bar (by default). In most cases you actually want the cursor to position to the first field in the "panel body", so the CURSOR_LOC parameter is required, even if all the fields on the panel are output only fields.

How does the program ever end?

This program has an apparently endless loop.

However, it can be ended by the Exit key, the Cancel key or by the user switching to the action bar and choosing to use some other function by selecting a choice from a pull down.

An accelerator key may be used as well, which is functionally identical to choosing from a pull down.

What program runs after this program?

That's up to the user controlling the application, and is of no concern to this particular program.

Action bar functions are small, robust and self contained. They do not ever trust (or try to control) what happens before or after they execute.

This is the "non-procedural" component of action bar programming. Just how the user can get from one function to another is not immediately apparent from the procedural components of the code.

Once you are comfortable with the concept that the "user drives the application" you should find that your programming tasks are actually much simpler than in a fully procedural system.

If they are not, or you are having conceptual problems with the way action bars work, or you find you are generating large amounts of code, please do not hesitate to contact your product vendor for assistance.

9.18.2 Sample Program: Displaying Details of an Object

EXCHANGE FIELDS(#OBJECTKEY) OPTION(*ALWAYS) GROUP_BY NAME(#PANEL) FIELDS(#OBJECTA01 ... #OBJECTANN)

```
BEGIN_LOOP
FETCH FIELDS(#PANEL) FROM_FILE(OBJECT)
WITH_KEY(#OBJECTKEY)
IF_STATUS IS_NOT(*OKAY)
MESSAGE 'Object to be displayed cannot be found .. .
select desired object'
MENU
ENDIF
DISPLAY FIELDS(#PANEL) IDENTIFY(*DESC)
CURSOR_LOC(*ATFIELD #OBJECTA01)
END_LOOP
```

Q & A

What does the program do?

It does only one thing. Allow the user to display the details of the object that they are working with.

Should it be enhanced to allow the user to change the object?

Probably not as Action bar programs should be designed to perform only one "action".

If I could have validated the "key" in the select object program I would not have had to code the "not found" check here?

No, that's not true. Even if the check could have been put into the object selection routine first, you would still need it in this program.

Some reasons: User invokes this program before selecting the object required, or, another user deletes the object between the "Select" and "Display" actions.

Overriding reason: without checking for the "not found" situation this program could not be called robust. Additionally, it is relying on another routine to be executed first. This is a clear violation of its self-containment and non-reliance on the pre or post execution of other routines.

What does the MENU command do?

It transfers control back to the process logical "menu", which in this application is actually being presented to the user as an action bar.

In other words, it cause the "action bar" to receive control and display the error message.

The user can then select the desired (or suggested) action from the action bar to correct the problem.

Does this mean more screen interactions when the user makes a keying error when typing in the "key" in the "Select" routine?

Yes it does.

There would be one more. The user would have to choose the "Select" option again and correct the "key" value.

However, the complexity of the application has to be played off against the chance of the error occurring and the degree of disruption the way it is handled causes.

In this case the chances are moderate, and the disruption is nil, so there is no real problem: just a different and simpler way of doing things.

Can I include lists on the screen panel as well?

Yes. This is just a simplistic generic example.
9.18.3 Sample Program: Changing an Object

```
EXCHANGE FIELDS(#OBJECTKEY) OPTION(*ALWAYS)
GROUP_BY NAME(#PANEL) FIELDS(#OBJECTA01 ... #OBJECTANN
    (#AB$TABFLD *INPUT *NOID))
BEGIN LOOP
 FETCH
         FIELDS(#PANEL) FROM FILE(OBJECT)
     WITH KEY(#OBJECTKEY)
 IF STATUS IS NOT(*OKAY)
 MESSAGE 'Object to be changed cannot be found ...
      select desired object'
 MENU
 ENDIF
MESSAGE 'Key in object changes and press Enter.'
 REQUEST FIELDS(#PANEL) IDENTIFY(*DESC)
     CURSOR_LOC(*ATFIELD #OBJECTA01)
         FIELDS(#PANEL) IN FILE(OBJECT)
 UPDATE
 MESSAGE 'Object has been updated with new details'
END LOOP
```

Q & A

What does the program do?

It does one and only one thing. Allow the user to change the details of the object that they are working with.

What is the AB\$TABFLD field for?

This example assumes that field AB\$TABFLD has been defined in the data dictionary with input and output attribute ND (non- display) only. You can put a field like this into your dictionary. Use any name you like, there is nothing special about the name AB\$TABFLD. Make it type alphanumeric, length 1. It appears on the input capable screen panel created by the REQUEST command as the last field on the screen. Since it has no identification, and attribute ND, the user cannot see it or anything that they type into it. It is effectively invisible to the user, other than for the fact that the cursor will tab into the field if they tab around the panel body / action bar.

It is positioned as the last field on the panel body to prevent the user from accidentally tabbing into the action bar when they modify the last visible field on the panel body (e.g.: OBJECTANN).

Imagine that field AB\$TABFLD was **not** on the screen panel.

If field OBJECTANN was numeric and had attribute FE (field exit required) or RB (right adjust and blank fill), then imagine what happens when the user changes the field's value.

They change the value and are forced to use the field exit key. Where does the cursor go now? Straight up to the first action bar option. What happens when they press enter? It is interpreted as a switch to action bar request because of the cursor location.

What the user really wanted to do was update the OBJECT file.

This is why AB\$TABFLD is used. In this scenario, the use of the field exit key in field OBJECTANN would tab the cursor into the invisible "dummy" field. When the user pressed enter, the cursor would still be in the panel body, so the users request would be interpreted correctly.

It is recommended that you put a field like AB\$TABFLD into the data dictionary, with the correct attributes and help text, and then use it in all situations like this.

Should this function be broken up into smaller functions?

Quite possibly. One of the advantages of action bar processing is that a larger array of "actions" can be presented to the user over what was often practical in menu driven systems.

For instance, the OBJECT being dealt with in this program might be an employee master file. So rather than have one conceptually large "Change Employee" pull down option / function, it might actually be broken down into a series of conceptually smaller pull down choices / functions such as "Change Address", "Change Salary", "Change Phone Numbers", etc, etc.

Often these smaller functions are more readily identified by users with their **workplace practices**, rather than the "Change Employee" type of function, which has a very subtle hint of data modeling in it (ie: understanding that address details are "attributes" of an "entity" called an employee).

9.18.4 Sample Program: All 3 Functions in One Program

So far we have 3 example action bar applications to "Select", "Display" and "Change" our "object".

The next step would be to arrange them into an action bar, or to plug them into a previously set up prototype action bar.

Both these subjects have been described in some detail in earlier sections.

Once this setup has been done, the action bar can be used.

However, there is one final step that might be considered to improve the performance of the action bar application.

If this example was run as is, the user would jump between 3 separate compiled RDML functions, and each jump would involve the opening and closing of files, deactivating and reactivating of programs, etc.

This overhead may be small and acceptable in most applications, but to maximize performance you can combine multiple action bar processing programs into one RDML function.

The secret to this performance trick is that the action bar control logic has some "intelligence" built in to it.

Normally, if a user switches to the action bar and selects another pulldown choice / function, the current function is terminated and the new one is started. This is what causes the overhead.

However, if the selected pull down choice is actually going to use the same function as the one that is currently active, the action bar logic just "restarts" the current function from the beginning.

No close/open or deactivate/reactivate logic is performed.

It should now be apparent why the PD\$OPT and AB\$OPT values are defined in the action bar control table. They are what allow a "combined" action bar function to work out what action it has to perform.

The AB\$OPT value indicates the action bar option that was selected by the user.

The PD\$OPT value indicates what pull down choice was selected by the user.

If our 3 example pull down choices "Select", "Display" and "Change" has PD\$OPT values of SEL, DIS and CHG respectively, then the obvious mainline for our combined RDML function would look something like this:

EXCHANGE FIELDS(#OBJECTKEY) OPTION(*ALWAYS)

```
CASE OF_FIELD(#PD$OPT)
WHEN VALUE_IS('= SEL')
EXECUTE SUBROUTINE(SEL_OBJECT)
WHEN VALUE_IS('= DIS')
EXECUTE SUBROUTINE(DIS_OBJECT)
WHEN VALUE_IS('= CHG')
EXECUTE SUBROUTINE(CHG_OBJECT)
ENDCASE
```

All that is required now is to combine our previous example functions into subroutines of this mainline like this:

SUBROUTINE NAME(SEL_OBJECT)
<< "Select" routine as per previous example >>
ENDROUTINE

SUBROUTINE NAME(DIS_OBJECT) << "Display" routine as per previous example >> ENDROUTINE

SUBROUTINE NAME(CHG_OBJECT) << "Change" routine as per previous example >> ENDROUTINE

Next we compile this new "combined" function and finally change the action bar control table to indicate that all the pull down choices should invoke this new "combined" function.

The **restart logic** is very simple.

When a DISPLAY or REQUEST command is logically terminated by the user from the action bar, and the new selected function is the same as the one that is currently active, control passes back to the first executable command in the program again (ie: the CASE command).

Lists are not cleared, variables are not re-initialized, etc.

It is just as if you executed a GOTO command to the start of the program.

Finally a **word of warning**: even though this function now contains 3 combined functions, their logical "separation" must be maintained.

Do **not** use the fact that these routines are now combined to start introducing interdependencies and cross executions between them.

Their independence must be maintained.

9.19 Action Bars: Another Book You Must Read

The **key component** of the CUA 1989 standards for action bar driven systems is that of the "Object-Action" approach to application design.

Please refer to the IBM supplied CUA manual *Common User Access:* LANSA Basic Interface Design Guide (SC26-4583) for more details of the Object-Action approach to design. Refer especially to the chapter *Action Bar and Pull-Downs* and the Appendix *Designing an Object-Action Orientated Application*.

10. Examples of using LANSA

Examples of LANSA coding taken from LANSA sites are presented in this chapter.

- 10.1 Handling Missing Records Simply
- 10.2 Text String Processing
- 10.3 Temporary Work Files
- 10.4 Page at a Time Scrolling
- 10.5 Left to Right Scrolling / "Subfile" Folding
- 10.6 Saving and Restoring Field Values
- **10.7 Validation Subroutines**

10.1 Handling Missing Records Simply

The following code is very common in reporting programs:

FETCH FIELDS(#PRODES) FROM_FILE(PROMST) WITH_KEY(#PRO IF_STATUS IS_NOT(*OKAY) CHANGE FIELD(#PRODES) TO("'N/AVAILABLE''') ENDIF

This effectively says, "if you can't find the product's description, print it as N/AVAILABLE".

HOWEVER, the following code will achieve exactly the same result and save 2 lines of coding:

CHANGE FIELD(#PRODES) TO("'N/AVAILABLE'')

```
FETCH FIELDS (#PRODES) FROM_FILE (PROMST) WITH_KEY (#PRO
```

If the FETCH fails to find any details, the product description field will remain unchanged.

10.2 Text String Processing

The LANSA RDML is quite good at string handling. When this is combined with the flexibility of true subroutine parameters, a very simple method of assembling, (and in this case printing), text strings results.

Consider the following subroutine:

```
SUBROUTINE NAME(PRINT TEXT) PARMS(#S01 #S02 #S03 #S04 #S05
            FIELD(#S01) TYPE(*CHAR) LENGTH(30)
   DEFINE
            FIELD(#S02) TYPE(*CHAR) LENGTH(30)
   DEFINE
            FIELD(#S03) TYPE(*CHAR) LENGTH(30)
   DEFINE
   DEFINE
            FIELD(#S04) TYPE(*CHAR) LENGTH(30)
            FIELD(#S05) TYPE(*CHAR) LENGTH(30)
   DEFINE
   DEFINE
            FIELD(#R01) TYPE(*CHAR) LENGTH(132)
   DEF LINE NAME(#R01L) FIELDS(#R01)
          BUILTIN(BCONCAT) WITH ARGS(#S01 #S02 #S03 #S04 #S05
   USE
        TO_GET(#R01)
   PRINT
           LINE(#R01L)
 ENDROUTINE
This subroutine can be used like this:
  EXECUTE PRINT_TEXT("Product" +
```

#PRODNO +
"'is out of stock. Please contact'"
#SUPPLIER +
"'to order more."")

to print:

Product 836 is out of stock. Please contact ACME ENGINEERING to order more.

Or like this:

```
EXECUTE PRINT_TEXT("'Your departmental supervisor'" +
#TITLE +
#NAME +
"'should be informed of this error'" +
*BLANKS)
```

to print:

Your departmental supervisor Ms Cullen should be informed of this error.

10.3 Temporary Work Files

The need for "temporary" files arises in nearly every application system at one time or another. Temporary files are commonly used for holding summarized data, passing large amounts of information between programs, or for preparing "flat" files for transfer to other systems (such as PCs).

The technique described in the following example is not new, but the application of it within LANSA yields some useful results.

Example

Given that you identify a need for a temporary file called WRKFILE which contains the fields A, B and C, the suggested technique is as follows:

1.Define fields A, B and C in the data dictionary (if not already defined).

2.Define file WRKFILE as a normal LANSA database file in the default data library of the current partition. Then make it operational.

Note that this version of the file will never actually contain any data.

3.When you want to use WRKFILE in an RDML program, code the following starting lines into the program:

```
CHANGE FIELD(#LIB) TO(*PARTDTALIB)
```

EXEC_OS400 COMMAND('CRTDUPOBJ OBJ(WRKFILE) FROMLIB(#L TOLIB(QTEMP)')

POINT FILE(WRKFILE) TO_LIBRARY(QTEMP)

This code will cause a duplicate version of file WRKFILE to be created in library QTEMP (your job's temporary library), and will then "point" all I/O commands involving file WRKFILE to the version of the file in QTEMP.

The best features of using work files in special library QTEMP are:

- Work files are unique to the job which created them, and will not be shared with any other users of the system.
- Work files and all the data in them are automatically deleted when the job which created them ends.

10.4 Page at a Time Scrolling

The following RDML program generically searches for employees by surname. The user specifies all or part of an employee's surname and a resulting list of employees with names matching the request is displayed.

```
******
         Define work variables and browse list to be used
         FIELD(#L1COUNT) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE
DEF LIST NAME(#L1) FIELDS((#SURNAME) (#GIVENAME) (#EMPNC
#ADDRESS1)) COUNTER(#L1COUNT)
******** 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)
         FIELDS(#L1) FROM FILE(PSLMSTV1)
SELECT
     WITH_KEY(#SURNAME) GENERIC(*YES)
ADD_ENTRY TO_LIST(#L1)
ENDSELECT
******
        If names found, display list to user
      COND('#L1COUNT *GT 0')
IF
          BROWSELIST(#L1)
DISPLAY
******
         else issue error indicating none found
ELSE
MESSAGE
           MSGTXT('No employees have a surname matching request')
```

ENDIF

********* Loop back and request next name t**o 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 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.

The following program is functionally identical to the first example, but produces the same results in a much shorter time and with less use of computer resources.

New or modified commands are indicated by -->.

```
******
           Define work variables and browse list to be used
           FIELD(#L1COUNT) TYPE(*DEC) LENGTH(7) DECIMALS(
  DEFINE
___
> DEFINE
           FIELD(#L1PAGE) TYPE(*DEC) LENGTH(7) DECIMALS(0)
            FIELD(#L1TOP) TYPE(*DEC) LENGTH(7) DECIMALS(0)
--> DEFINE
            FIELD(#L1POS) TYPE(*CHAR) LENGTH(7)
--> DEFINE
> DEF_LIST NAME(#L1) FIELDS((#SURNAME) (#GIVENAME) (#EMP)
       ESS1)) COUNTER(#L1COUNT) PAGE SIZE(#L1PAGE) TOP EN
       L1TOP) SCROLL_TXT(#L1POS)
  ******** 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$KEY) TO(UP)
--> CHANGE
             FIELD(#L1TOP) TO(1)
            FIELDS(#L1) FROM_FILE(PSLMSTV1)
--> SELECT
       WITH KEY(#SURNAME) GENERIC(*YES)
        WHERE('#IO$KEY = UP') OPTIONS(*ENDWHERE)
-->
            SUBROUTINE(DISPLAY) WITH_PARMS("'More...'")
--> EXECUTE
 ADD_ENTRY TO_LIST(#L1)
  ENDSELECT
  ******
           If names found, display list to user
       COND('#L1COUNT *GT 0')
  IF
             SUBROUTINE(DISPLAY) WITH_PARMS("Bottom")
--> EXECUTE
  ******
           else issue error indicating none found
  ELSE
             MSGTXT('No employees have a surname matching request')
  MESSAGE
  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(5)
    --> CHANGE FIELD(#L1REMN) TO('#L1COUNT / #L1PAGE')
```

> IF COND('(#L1COUNT *NE 0) *AND (#IO\$KEY = UP) *AND ((#L1)
S = "Bottom") *OR (#L1REMN *EQ 0.00000))')

--

```
> DOUNTIL 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 used 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 should then be set up as an Application Template so that it is accessible to all RDML programmers.

10.5 Left to Right Scrolling / "Subfile" Folding

Frequently users request access to the IBM i SFLFOLD (subfile fold) facility which LANSA does not support.

However, by using a simple left-right scrolling facility the technique can be emulated, and often made better.

Imagine that a "list" of information that is 3 screens wide has to be presented to the user.

Scrolling up and down the list is no problem, however, scrolling from left to right can also be achieved like this:

```
def_list #list1 fields(whatever) top_entry(#top)
def_list #list2 fields(whatever) top_entry(#top)
def_list #list3 fields(whatever) top_entry(#top)
select <whatever>
add entry #list1
add_entry #list2
add_entry #list3
endselect
change #listno 1
dountil '#io$key = RA'
 case #listno
 when = 1
 display browselist(#list1) userkeys((8 'Right'))
 when = 2
 display browselist(#list2) userkeys((7 'Left')(8 'Right'))
 when = 3
 display browselist(#list3) userkeys((7 'Left'))
 endcase
 case #IO$key
 when = 07
 change #listno (#listno - 1)
 when = 08
```

change #listno (#listno + 1) endcase

enduntil

The user can then use function key 7 (scroll left) or function key 8 (scroll right) to scroll left and right across the lists. The use of the same top-entry value for all 3 lists should ensure that the same page of the list is always displayed.

10.6 Saving and Restoring Field Values

Sometimes programmers have a need to "save" the current values of fields and then restore then at some later time.

For instance a user may need to save the values of fields #A -> #C, and to do this you often code fields like this:

define #sava reffld(#a)
define #savb reffld(#b)
define #savc reffld(#c)

and later (to save the values)

change #sava #a change #savb #b change #savc #c

and still later (to restore the values)

change #a #sava change #b #savb change #c #savc

An easier way to do this is to use a "working" list as a "stack" like this:

def_list #stack (#a #b #c) type(*working) entrys(1)

to save the current values, "stack" them like this:

inz_list #stack

to restore the values, "unstack" them like this:

get_entry 1 #stack

Some other points about this technique are:

- You can have as many "stacks" as you like.
- The same field can be nominated in different stacks.
- The resulting code is probably more efficient than the original
- The stack can have more than 1 entry, allowing it to "stack" and "unstack" multiple different values. This is just like a true "stack" with "push" and "pull" facilities.
- The "AS/400 Work With" style template uses this technique to save and restore the field key values that the user originally keyed to build a list of

objects to work with.

10.7 Validation Subroutines

Some programmers like to code validation logic into subroutines, and do not easily accept the LANSA approach of:

```
request <<data>>
begincheck
<<validate>>
endcheck
```

Unable to resist the urge to use a subroutine they soon change the program to be like this:

```
dountil #errors = n
request <<data>>
execute validate
enduntil
```

```
subroutine validate
begincheck
<<validate>>
if_error
change #errors Y
else
change #errors N
endif
endcheck
endroutine
```

and are surprised when the whole program aborts

The reason is that the ENDCHECK command, by default, returns control to the last display (if it can be found in same routine), otherwise it aborts the program. In this subroutine the ENDCHECK command has no last display to return control to, so it aborts. If you must code validation routines, the best structure is probably something like the following:

```
dountil #errors = 0
request <<data>>
execute validate
enduntil
```

subroutine validate
change #errors 0
begincheck keep_count(#errors)
<<validate>>
endcheck if_error(*next)
endroutine

11. Application Performance on IBM i

11.1 Macro Performance Factors and Micro Performance Factors	11.16 Some Programming Considerations
11.2 Interactive, Batch and Overnight	11.17 IBM i Authority Checking
Processing	11.18 End User And Developer
11.3 Using Working Lists and the	Machines
KEEP_LAST Parameter	11.19 Team Sizes
11.4 Keeping Heavily Used Files	11.20 Education
Open	11.21 Attitude and Commitment
11.5 Using the OPEN Command	11.22 Work Practices
Data	11.23 Unlearning
11.6 Other File Opening	11.24 Loss of 3GL Skills
Considerations	11.25 Signs of Trouble
11.7 The FUNCTION Command	11.26 Large Numbers - Approach with
11.8 Clean Up Considerations for	Caution
Heavy Usage Functions	11.27 Estimating Application
11.9 Using *DBOPTIMIZE /	Performance by Running an Application
*DBOPTIMISE	11.28 Estimating Application
11.10 PAG Trimming	Performance by Calculating an ERUF
11.11 Execution and Security Settings	11.29 Breakdown of RDML Commands into Resource Groups
11.12 People and Machine	11.30 Breakdown of Commonly Used
Performance - Some Things to	Built-in Functions into Resource Groups
Consider	
11.13 Co-ordinate System Settings	
11.14 Some Database Design Considerations	
11.15 Some Functional Design	

Considerations

11.1 Macro Performance Factors and Micro Performance Factors

Anything that affects the performance on IBM i will usually fall into one of the following categories:

• Macro performance factors: these are factors that each have a very large impact on the way a machine performs.

Some examples of macro performance factors include:

- Interactive vs. Batch job priorities
- Number of batch jobs executing concurrently
- Micro performance factors: these are factors that individually have very small impact on the overall way a machine performs.

Some examples of micro performance factors include:

- Activity levels, pool sizes, etc.
- Number of access paths being maintained
- Number of online users
- Amount of remote communications being performed

Generally speaking, if you have badly set macro performance factors, your machine will run very badly - but it can be easily and cheaply changed to run much better.

Changes made to an individual macro performance factor will usually produce a large and immediate improvement.

With micro performance factors the reverse is often true. If you have a lot of badly set micro performance factors, they will tend to "accumulate", causing your machine to run badly. It may be very difficult and/or expensive to change these factors to improve your machine performance.

Changes made to an individual micro performance factor will often produce only very slight improvements in performance.

11.2 Interactive, Batch and Overnight Processing

To ensure smooth running, it is recommended that all development and "end" users of the system conform to the following guidelines:

- Any work that can be done in batch should be done in batch.
- Any batch work not required today should be submitted for overnight execution.

To this end, it is recommended that you think about configuring your system to run batch work along these lines:



- Daytime batch jobs that are reasonably expected to take less than five minutes to execute should be sent to job queue 1 for execution.
- Any other daytime batch job should be sent to job queue 2.
- Jobs that are not required on the current day may be submitted to job queue 3. This job queue is held during the day and only released at night.

Some other points about running batch jobs on the IBM i are:

- Resist the temptation to allow more than two batch jobs to execute concurrently (no matter how large your machine is), until all other performance factors have been fully evaluated and tested.
- When evaluating batch throughput, measure the time from when the batch job was placed onto the job queue until it completes NOT from the time is begins to execute until it completes.
- Think about leaving your machine running 24 hours a day, or at least running at night until all batch work has been completed. Either of these approaches can be fully automated and usually do not require an operator to be in attendance.

11.3 Using Working Lists and the KEEP_LAST Parameter

LANSA provides two main facilities which can be used to reduce the number of program database accesses. These are the data construct called a "working list" and the KEEP_LAST parameter of the FETCH command.

The use of working lists for high performance summary reporting has already been discussed in previous sections. However, this is not their only use. Working lists can also be used to minimize database I/Os as they can be "looked up" much faster than a database I/O can be performed. Examples of this latter use of working files can be found in the LANSA Technical Reference.

In situations where complex look up criteria or multiple files are involved, the working list is a viable solution for database access reduction. HOWEVER, in most simple situations, the KEEP_LAST parameter of the FETCH command can produce the same results with much less effort.

In the following example, the KEEP_LAST parameter of the FETCH command is used in a program which reads and prints details of all general ledger transactions from a file called GLTRANS.

In this example, associated with each transaction is a company number (#COMPNO), and the actual company name (#COMPNAME, from file COMPANY) must appear on the report:

```
DEF_LINE NAME(#REPORTLIN) FIELDS(#TRANSNUM #TRANSTYP #
#COMPNO #COMPNAME)
```

```
SELECT FIELDS(#REPORTLIN) FROM_FILE(GLTRANS)
FETCH FIELDS(#COMPNAME) FROM_FILE(COMPANY) WITH_KEY
PRINT LINE(#REPORTLIN)
ENDSELECT
ENDPRINT
```

If there were 10,000 transactions in GLTRANS this program would perform exactly 20,000 database accesses.

However, if the FETCH command was modified:

```
FETCH FIELDS(#COMPNAME) FROM_FILE(COMPANY) WITH_KEY
KEEP_LAST(15)
```

and there were only 15 companies, then the program would now perform at most 10,015 database accesses, and thus run in about half the time required by the original version.

Note: If there are more than 15 companies the program will not fail, it will simply do the extra I/Os required. Refer to the FETCH command in the *LANSA Technical Reference* for more details of the KEEP_LAST parameter before attempting to use it.

Another example of the KEEP_LAST parameter is the improvement of a widely used technique of database access minimization. For instance, very often programmers code logic like this:

IF COND('#PRODNO *NE #PRODNOLST') FETCH FIELDS(#PRODES) FROM_FILE(PROMST) WITH_KEY(#PROI CHANGE FIELD(#PRODNOLST) TO(#PRODNO) ENDIF

which is basically saying: "get the product description, if the product being processed is different to the last one processed". It is very effective.

However, by using the KEEP_LAST parameter the coding required to achieve the same performance benefit can be reduced to just one line - and there is no need to use a work field (like #PRODNOLST) to keep track of the last value processed.

FETCH FIELDS(#PRODES) FROM_FILE(PROMST) WITH_KEY(#PROI KEEP_LAST(1)

11.4 Keeping Heavily Used Files Open

The LANSA OPEN command can be used to keep file(s) open until a specific CLOSE command is issued against them.

For instance, imagine a small order processing system that could be visualized like this:



When a user enters this system and begins jumping from ORD001 to ORD002 to ORD003, and so on, a fair amount of computer resource can be wasted opening and closing files used by all the functions.

For instance, all 3 functions might use files TABLES (system tables), ORDHDR (order header details) and ORDLIN (order line details).

Thus when the user exits from ORD001 all 3 files are closed. When the user invokes ORD002, all 3 files are opened again. This continual opening and closing is a waste of computer resource and degrades response times.

To fix this problem we might code a "file opener" called ORD004 that looked like this:

```
OPEN FILE(TABLES) USE_OPTION(*KEEPOPEN)
OPEN FILE(ORDHDR) USE_OPTION(*KEEPOPEN)
OPEN FILE(ORDLIN) USE_OPTION(*KEEPOPEN)
CALL PROCESS(ORDWRK)
```

which would fit into the existing system like this:



In this situation, the implicit OPEN and CLOSE requests issued by ORD001, ORD002 and ORD003 are ignored, which results in much faster processing.

11.5 Using the OPEN Command Instead of a Logical View to Order Data

The use of the LANSA OPEN command with the parameter USE_OPTION(*OPNQRYF) to invoke the operating system command OPNQRYF (open query file) has already been discussed in relation to the production of flexible reporting programs.

This section describes how the KEYFLD parameter can be used to alter the order in which data is processed/printed by a report program.

Another aspect of this facility relates to total system performance.

When you are going to produce a report which must be ordered in a way that does not reflect the key of any existing logical file, you basically have 2 choices:

- The first is to create a new logical file which enables the program to view the data in the way required to produce the new report.
- The second is to use the OPEN USE_OPTION(*OPNQRYF) command to create a temporary access path which will enable you to view the data in the required order.

If the report is not commonly used, and there are less than 100,000 records in the file, the second is probably the preferred alternative, mainly because:

- Creating a logical file with an immediately maintained access path puts additional load on all system users.
- Even if the file has a deferred or rebuild style access path it will still mean continual system overheads as well as causing LANSA to create a larger I/O module for the file.
- An additional logical view becomes an additional burden whenever a system is being changed or maintained. It must be understood and accounted for by the person changing the system.

11.6 Other File Opening Considerations

By default, all RDML functions attempt to "overlap" the first user/screen interaction with a mass opening of all the databases files which will be used by the function. In other words, while the user is working with the first screen, a concurrent open of the database is being performed. Effectively, this technique is designed to reduce the user's perceived response time.

Although this approach always works, in some situations it does not perform the required action in the best possible way. For example, some of the situations where this action may not be the best result are:

• When not all the files in the program need to be constantly open.

A classic example of this is the report programs described in the previous section. Structurally, these report programs look something like this:

```
<< if running online >>
```

```
get report parameters and submit report job
```

<< else >>

```
produce required report using report parameters
```

- << endif >>
- In the above case, obviously no database files whatsoever are required when the job is running online. Only when the job is running in batch and the report is being produced are the database files needed.
- Performance of the online portion of this program (and any other program that does not need to keep open all its files) may be considerably improved by inserting this command which states that database files should only be opened when they are used (ie: on demand):

OPEN FILE(*ALL) USE_OPTION(*ONDEMAND)

• When the program is communicating to devices via a slow communications link.

Since LANSA attempts to overlap file opens with the first screen I/O it cannot effectively utilize the DFRWRT (defer write) parameter of the CRTDSPF (Create display file) command.

This parameter and command are part of the IBM i operating system. You

should refer to the appropriate IBM supplied manual for exact details of this command and the impact of the DFRWRT parameter on remote communication performance.

If an RDML function contains the command:

OPEN FILE(*ALL) USE_OPTION(*ONDEMAND)

then the RDML compiler will decide that there is never any need to overlap file opens with the first screen I/O, and will thus use DFRWRT(*YES) when creating the associated screen display file. This should improve perceived response time for remote users and reduce the overall system load of the program.

11.7 The FUNCTION Command

The FUNCTION command is specifically designed to allow some aspects of a compiled RDML program's logic to be altered for optimal performance in specific environmental conditions.

The OPTIONS parameter of the FUNCTION command allows the following values to be specified:

*NOMESSAGES

Specifies that the program will not receive from its caller, or route back to its caller, any messages (except if it fails). Using this option can significantly improve the entrance and exit resource use of an RDML program.

Mainly used in heavily used online applications and any form of subroutine, especially those working in a batch environment.

*DEFERWRITE

Specifies that any IBM i display file created to service DISPLAY, REQUEST or POP_UP commands should use the DFRWRT(*YES) parameter.

This value can significantly improve screen handling, reduce screen flickering and produce a better response time (particularly in applications run from remote devices). Refer to *Flickering Windows and Program Efficiency for Pop-Up Windows* for more details on the significance of the DFRWRT(*YES) parameter on the IBM i display files.

A good choice for most online applications where an "overlapped" database open is not required. Essential in applications that use the POP_UP command.

*HEAVYUSAGE / *LIGHTUSAGE

When a process is defined, an option to specify usage HEAVY or LIGHT can be nominated. This governs whether all the functions associated with the process "shutdown" when they have completed execution (LIGHT usage), or stay "active" (HEAVY usage).

This option allows the HEAVY or LIGHT characteristics of a particular function to be individually specified, regardless of what option the associated process uses.

*HEAVYUSAGE is a good choice for heavily used subroutines, particularly those working in a batch environment. Refer to 11.8 Clean Up Considerations for Heavy Usage Functions.

*DBOPTIMIZE / *DBOPTIMISE

This option indicates that "database access optimization" should be used. It is discussed in detail in the Technical Reference.

***DIRECT**

This option indicates that the function is eligible for potential direct calling. A direct call is considerably faster than a call via a process controller. Refer to the CALL command in the *LANSA Technical Reference* for a more complete explanation of how direct calls are used to improve performance.

11.8 Clean Up Considerations for Heavy Usage Functions

A function's usage may be set in one of these ways:

- The Anticipated Usage of the process it is in.
- It contains a FUNCTION RDML command with OPTION *HEAVYUSAGE or *LIGHTUSAGE
- It executes the SET_FOR_HEAVYUSAGE or SET_FOR_LIGHTUSAGE Built-In Function

Note that each successive way overrides the previous ways. That is, a function may be in a LIGHT usage process but contains FUNCTION OPTIONS(... *HEAVYUSAGE ...) – it will be heavy usage. Likewise a function may contain FUNCTION OPTIONS(... *LIGHTUSAGE ...) but have executed the Built-In Function SET_FOR_HEAVYUSAGE – it will be heavy usage.

In general, the way to clean up the resources used by a heavy usage function is to call it again and have it execute the SET_FOR_LIGHT_USAGE Built-In Function. However this is not always possible, given the type of function that it may be.

- Functions CALLed or activated via CALL_SERVER_FUNCTION Built-In Function may use this mechanism to clean up. One suggestion would be to create a field for this and only exchange it when the heavy usage function is to clean up.
- Trigger functions could be coded to execute the SET_FOR_LIGHT_USAGE Built-In Function when called for the After Close event.
- RDML Built-In Functions can use this mechanism in their shutdown logic.
- The following functions cannot use this mechanism:
 - Prompting functions.
 - System Variable Evaluation functions.
 - Complex logic functions.

11.9 Using *DBOPTIMIZE / *DBOPTIMISE

The special *DBOPTIMIZE option can simply and quickly improve the performance of any I/O bound application program in almost any situation.

Do not attempt to use *DBOPTIMIZE until you understand all the implications of using it. For details, refer to Using *DBOPTIMIZE / *DBOPTIMIZE_Batch in the LANSA for iSeries User Guide.

11.10 PAG Trimming

It is possible to trim the PAG being used in large online environments by preinvoking the LANSA supplied system variable evaluation programs.

To invoke this facility place a call to M@SYSVAR and F@SYSVAR very early in the invocation stack. For example, a call during sign on to the IBM i may be considered. Such calls are extremely fast and do not occupy any significant PAG space. What is important is that the "early" or "pre-invocation" call ensures that the little PAG space they use is placed very early in the total PAG space, thus reducing PAG fragmentation when subsequent callers invoke either of these routines and then end.

To place a "pre-invocation" call to M@SYSVAR or F@SYSVAR simply use a normal CALL operation passing a character(1) parameter containing binary zeros (ie: X'00' in CL or *LOVAL in RPG/400)

To place a "close down" call to M@SYSVAR or F@SYSVAR simply use a normal CALL operation passing a character(1) parameter containing binary ones (ie: X'FF' in CL or *HIVAL in RPG/400)

11.11 Execution and Security Settings

The following topics describe the system flags in the IBM i data areas that are set via the Work with Execution and Security Settings screen:

- 11.11.1 Setting the System Run Time Locking Setting Off
- 11.11.2 Turning Function Level Security Checking Off
- 11.11.3 Turning All Process and Function Security Checking Off
- 11.11.4 Turning All File Access Security Checking Off

If you change one of these flag settings, you should also change it across all your separate LANSA systems.

An "audit" of the settings of all your data area settings across all your LANSA systems, is a useful check to perform from time to time.

In the *LANSA for iSeries User Guide* is the complete layout of the System Definition Data Area DC@A01 in which the following settings are located.

11.11.1 Setting the System Run Time Locking Setting Off

If a data area flag field called *Execution locking on process* is set to "Y", a shared lock will be set on the process during execution. If a shared lock on the process cannot be established then execution of the process will fail.

This data area flag is changed on the Work with Execution and Security Settings described in the *LANSA for iSeries User Guide*.

This means that every time a process is invoked (or a function it contains is invoked) a shared lock is issued against a data area with the same name as the process. Whenever the process (or function) completes execution the shared lock is released.

This locking feature is provided to ensure that you cannot delete or re-compile a process or function while someone else is using it.

Performing this lock and unlock logic takes time and uses computer resource. If desired it can be disabled by setting the run time lock flag to N (for no). However, once disabled, the possibility of deleting an object that someone else is using arises. If this happens, their job will crash.

Note: Compiled processes (not compiled RDML functions) are sensitive to the run time lock flag setting at the time of compilation. If you later change the flag's value you may have to recompile all compiled processes before being able to achieve any performance benefit.

Also see

11.13 Co-ordinate System Settings

11.11.2 Turning Function Level Security Checking Off

If a data area flag called *Function level security* is set to "Y", LANSA function level security is always enabled. This means that every time a process menu is displayed a check must be made against all associated functions and attached processes / functions to determine if the user is authorized to use them.

This data area flag is changed on the Work with Execution and Security Settings described in the LANSA for iSeries User Guide.

If the user is not authorized, the options involved do not appear on the process menu.

While this facility provides a simple and easy way of building variable menus by user, it requires system resources to perform the security checks.

If you have no need to build user variable process menus, consider setting the system flag to "N" to indicate that function level security is not required.

Also see

11.13 Co-ordinate System Settings
11.11.3 Turning All Process and Function Security Checking Off

The flag that controls this option is the *Disable end user process and function security* which you set in the Work with Execution and Security Settings described in the *LANSA for iSeries User Guide*.

A "Y" in this flag indicates that run time process and function level security checking should be suppressed.

Any other value indicates that run time process and function security checking should be performed as normal.

Setting of this flag to "Y" may be appropriate for users that use other/external menu systems to control and secure access to LANSA processes and functions.

Minor performance improvements can be expected by setting this flag to "Y".

This flag is interpreted dynamically.

The setting of this flag has no effect on development activities involving the process or function definitions. They are still subject to normal security checking.

Also see

11.13 Co-ordinate System Settings

11.11.4 Turning All File Access Security Checking Off

The flag that controls this option is the *Disable end user file level security* which you set in the Work with Execution and Security Settings described in the *LANSA for iSeries User Guide*.

A "Y" in this flag indicates that run time file level security checking should be suppressed. Any other value indicates that run time file security checking should be performed as normal.

Setting of this flag to "Y" may be appropriate for users that use other/external menu systems to control/secure access to LANSA processes - and/or use the philosophy "if the program appears on your menu, then you should be able to access the files that the program requires".

Minor to moderate performance improvements can be expected by setting this flag to "Y", especially in *DBOPTIMIZE functions.

This flag is interpreted dynamically.

Normal (ie: non-*DBOPTIMIZE) functions do not need to be recompiled to react to this new flag.

The setting of this flag has no effect on development activities involving the file definitions. They are still subject to normal security checking.

Also see

11.13 Co-ordinate System Settings

11.12 People and Machine Performance - Some Things to Consider

The following topics lists, in no particular sequence, a variety of points that involve the performance of people and machines when they are using 4GL tools on the IBM i.

Most of these points have been supplied by existing LANSA users from their own experiences, or noted by the developers of LANSA when observing the types of problems reported when developing applications using 3GL or 4GL techniques.

The list is presented in an abbreviated point form.

If a particular point is of interest, further reading of LANSA or IBM manuals will be required.

This list is neither complete, nor definitive.

11.13 Co-ordinate System Settings

• The setting of the system flags listed in 11.11 Execution and Security Settings should be checked and made the same across all LANSA systems in the environment.

11.14 Some Database Design Considerations

Logical data modeling requires a different mindset to traditional database design. The training of people to understand logical data modeling is essential.

MIS staff with limited program coding experience cannot be expected to take on the data modeling component of a project without the necessary training and time to fully develop their experience and skills in small test cases.

Never commence your first data modeling project on a large system. Gain skills and experience in small systems and test cases before attempting to take on a large system.

Minimize the number of logical views. File index maintenance is one of the largest machine resource users. What appears to be one I/O in your program may actually involve many hundreds of disk accesses. Investigate OPNQRYF for rarely used access paths.

Investigate using the IBM i High Speed Table facility to allow faster access to read only validation/decode tables. For more information refer to Files in the *LANSA for iSeries User Guide*.

11.15 Some Functional Design Considerations

The speed of an application is a simple function of its complexity and how many other jobs are concurrently executing in the system.

Unrealistic expectations of hardware performance may be a problem. Insufficient CPU/DASD power cannot possibly support the application complexity and number of users.

Quality and speed have a price. If you want these, then you must be prepared to pay a reasonable price.

Remember KISS - Keep it Short and Simple. Avoid "over-development" by resisting the urge to give users 150% of what they want, presented to them in a fashion that is 200% better than before.

These types of solutions impress end users, but usually add to the size and the complexity of the application (which increases its maintenance cost) and the amount of resource it consumes (which may mean that you need a more powerful computer to run it).

If you progressively alter your original STANDARD intention into a SUPER solution, you must make sure that you can afford the "running costs" before you implement it.

This may be a problem when using 4GL tools.

Since 4GL tools are fast at designing and developing applications, they allow you to develop a SUPER application at the same cost, and in the same time frame, as you formerly produced a STANDARD application.

However, you must make sure MIS staff and machines are capable of running and maintaining the SUPER solution.

4GL tool users generally fall into 2 categories:

- Those that make better designed solutions, of similar complexity and functionality to their existing systems, in a much shorter time frame.
- Those that make better designed solutions, of much more complexity and functionality to their existing systems, in about the same time frame.

Both users gain significant benefits for their organizations, but the second type of users need to understand that the increase in the complexity and functionality may require more resources to run and maintain their applications.

11.16 Some Programming Considerations

Use *DEFERWRITE in all functions.

Use *NOMESSAGES whenever the function is not required to (re)route messages.

Use *HEAVYUSAGE when the function is used constantly.

Use *DBOPTIMIZE when the function is used constantly, or is a batch job that processes a large number of database records.

When using *DBOPTIMIZE ensure all files have SHARE(*YES).

Understand the difference between causing the file to be opened for shared access, and having the file defined for shared access.

Understand the difference between a shared open data path and a shared access path.

Use OPTION(*BLOCKnnn) on full file SELECT commands in batch.

Use KEEP_LAST(n) on all repetitive FETCH commands.

Use compiled processes at all times in production systems. Watch for programs called DC@P5001nn (where nn is the range 01 - 09) in the invocation stack. This indicates an interpretive mode process is running.

Use FUNCTION OPTIONS(*DIRECT) in all functions.

Use CALL PROCESS(*DIRECT) FUNCTION(ffff) whenever possible.

Use process name *DIRECT when nominating prompting programs in the data dictionary.

Check open files in job stacks for same file open multiple times for same use (e.g.: input). This may indicate share(*yes) is not being used or has not been implemented correctly. The same file open for different uses is acceptable.

Use the LANSA supplied menu system whenever practical. Avoid the constant in and out from user defined menu systems or CL menus.

Keep invocation stacks simple. Program A calls B calls C calls D calls E calls F type of logic is not good for performance on the IBM i, no matter what development tool is used.

Avoid using OPNQRYF on large files in online programs.

Avoid using more than 20 files in any one program. Avoid having more than 40 - 50 files open at any one time in a job.

Run batch jobs in overnight mode unless actually required. Do not run too many

batch jobs concurrently. Batch job I/O requests will always impact online job performance, no matter what scheduling and priority options are used for the CPU cycles.

11.17 IBM i Authority Checking

IBM i authority checking can impact system performance. The IBM i performance reports have a section that lists "exceptions".

In this list are "security" or "authority" exceptions. The rate at which these exceptions occur and impact system performance is shown in the following figure:

Authority Except	ions Percentage of Total CPU Cycles consumed
Rate per Second	B10 B20 B30 B40 B50 B60 B70
10	6 3 5 3 2 1 1
25	15 9 12 8 5 3 2
50	29 17 25 16 10 6 4
100	58 34 49 32 20 12 9
200	69 99 63 39 24 17
300	95 59 36 27
400	78 48 35
500	98 60 44
600	71 53
700	83 62
800	95 71
900	80
1000	89

If high exception rates are found, steps can be taken to alter IBM i security settings to reduce the number of exceptions that are occurring.

11.18 End User And Developer Machines

The load on an IBM i machine that mixes developers and end users is unpredictable and extremely variable over a short time period.

This type of environment is almost impossible to tune and is almost always subject to poor performance at one time or another.

The advent of the cheaper low end IBM i as developer machines is an attractive and cost effective way of stabilizing the end user environment and isolating the unpredictable developer environment.

11.19 Team Sizes

LANSA development appears to work best with small "teams" of up to 5 people who deeply understand the area of the project they are working on.

If a deadline is approaching, loading extra new people onto a project in an attempt to speed up development will almost always result in a slowing down of the project.

11.20 Education

Although the RDML component of LANSA is relatively easy to learn and use, it does not mean that someone can become an expert without ever having completed a training course.

Additionally, there are many more parts to the development of a complete application than just programming. The Systems Designer(s) must be experienced in database and application design methods that are compatible with 4GL tools, or else have the time available to become familiar with them.

Commencing a 4GL application development without proper education and training is likely to take longer than using existing 3GL techniques.

Training and education involve a lot more than learning how to write programs using the 4GL tool. The construction of application programs is a small part of the complete development cycle.

If you view the introduction of a 4GL tool as simply requiring the retraining of programmers to work in another "language", you will probably not achieve the results you expected.

11.21 Attitude and Commitment

The key success factors in using 4GL tools seem to revolve around the words "commitment" and "attitude".

Unless all members of the team are committed to the new technologies, and the methods they use, the result will almost invariably fail to meet expectations.

Additionally, they must have a positive attitude. This problem is most commonly found among long time 3GL programmers, who are not really interested in, or committed to, new 4GL technologies.

These two factors are probably the most critical in the success or failure of any application development and should never be under estimated.

11.22 Work Practices

Most 4GL products use much more machine resource during application development than 3GL development techniques do.

This is because the machine is doing much of the work that was formerly done on paper, or done by hand coding.

This probably means that an adjustment of existing work practices is required. In LANSA specifically, some of the adjusted work practices might include:

- If your machine already runs slowly, introducing a 4GL tool will make it worse. Be prepared to invest in hardware as well as software to support a better development cycle.
- Be realistic and have a development environment large enough to support the number of developers that are going to be used.
- If your application design efforts already suffer from lack of training and skills, introducing a 4GL tool without investing in proper training and education will only help to make a mess faster than was previously possible.
- Be prepared to invest time and money in education. Some of this will directly involve LANSA, but some of it may involve the acquisition of general data and application design skills.
- Use small, skilled project teams.
- Do not run multiple concurrent compilations on small machines.
- Minimize compiles by the extensive use of application templates.
- Paint final screen and report layouts after the application is functionally complete and tested.
- Do not run the online full function checker on functions of more than 500 lines (except to paint screen panels).
- When something is not required immediately, schedule it for compilation in off peak hours.
- Use application templates for all standard transactions. This will enforce your standards and minimize error rates.
- Invest the time and effort required to set up a good set of application development standards.
- Invest the time and effort required to set up a good set of application templates.

- Allow time for the implementation and distribution of new systems. This can be quite a lengthy process and many developers fail to allow for it in their schedules.
- Allow time for the proper training of end users.
- Use small programs. If you are continually creating programs of more than 300 lines, then something is going wrong. Large programs take much longer to compile, and this will in turn irritate other members of the development team.
- Do not try to solve problems with 3GL approaches.
- Never translate 3GL code into 4GL code.
- Never translate 3GL "logic" into 4GL "logic".
- Get standard program structures from templates. This way you are gaining from the experience of more skilled developers.

11.23 Unlearning

The methods used in data modeling and the new techniques available in application dialogue design mean that sometimes people have to "unlearn" or "discard" techniques they are already using.

The problem here is that some people are reluctant to do this and spend much of their time trying to "bend" the 4GL methodology to do things the way they always have done.

In most flexible 4GL products they can do this (to varying degrees), but the result is that often up to 70% of the development time is spent bending and fighting with the natural architecture built into the 4GL product.

This continual "bending" and "fighting" of the 4GL methodology may be the result of one or more of the following:

- Poor knowledge of new techniques that 4GL makes available.
- Resistance to change.
- Fear of losing 3GL skills.
- Designer wants to emulate existing systems.
- End users have forced the designer to "emulate" existing systems.
- Individual or corporate ego-based dialogue designs.

11.24 Loss of 3GL Skills

Some developers are reluctant to become involved with 4GL products, or develop a negative attitude to them because they fear they will lose their 3GL level design and coding skills.

Consider the following:

- "Top Gun" RPG programmers should realize that RPG has hardly changed in the last 8 10 years. Minor improvements like "read previous equal" and some new string handling options have appeared, and maybe we will see some "free format" options further down the track. While these new facilities are useful, they do not really offer any significant benefits to aid in system development.
- 4GL is inevitable. There will always be a need for 3GL programmers in maintenance, and specialized areas such as data communications. However, the role of the 3GL programmer in general commercial programming will continue to diminish, just like the role of machine level programmers has almost disappeared over the last 20 years.

This diminishing role is occurring for 2 main reasons. The first is that 4GL products are now flexible enough to deliver real systems with little 3GL intervention. The other is the very high cost of maintenance associated with existing 3GL systems.

• Design skills are more valuable than 3GL coding skills. The most valuable people in today's job markets are those with "high end" data and application design skills as well as project management skills. A broad exposure to 4GL products and technology is now essential to developing skills in these areas.

11.25 Signs of Trouble

Many of the previous points mentioned problems that may slow down the development of application systems. This section describes some specific signs that indicate a problem is occurring in one or more or the previous areas:

• Large teams, continually adding more people to the team.

A classic sign of an approaching deadline. The root cause of the problem is most likely to be many or all of the points previously mentioned acting together.

• Too many files in database.

Over normalized database. Lack of experience with database design has caused a completely normalized database (straight from logical model). Concessions to performance may be required, unless a commitment to sufficient hardware is made in early stages of design.

• Too many fields in files.

Under-normalized database. Too many concessions to application performance. No knowledge of relational techniques. Emulation of existing systems is also a problem here. A balance between hardware costs and maintainability of applications is required.

• Too many files open in application.

Result of over-normalized design in use, OR, too much complexity in individual applications. Again, this requires a balance between the design and the available hardware.

If you are using a system that is operating near to this "knee of the curve" you will find that a relatively small increase in the machine load will dramatically degrade the response time of all online users and batch jobs.

If you are using a system that is operating near to this "knee of the curve" you will find that a relatively small increase in the machine load will dramatically degrade the response time of all online users and batch jobs.

• Too many logical views.

Probably the result of an under-normalized design. If you are continually using more than 5 views per physical file then the underlying database probably has no normalized components. For rarely used views, use OPNQRYF or SQL.

• Continually creating large programs.

Symptom of many problems. Some may include: 3GL emulation, 3GL translation, 3GL coding, 3GL logic, "fighting the natural architecture", "ego based" dialogue designs, "ego based" programming techniques, retrofitting of major design changes, not using CUA, not using templates, validation rules in programs, poor knowledge of RDML facilities (lists, subroutines, etc.). Biggest problem is most likely to be a designer insisting on his/her own dialogue design, rather than using CUA and standard transaction resulting from application templates (shipped or in-house). This may be hard to overcome because of the "creativity" aspect.

• "Second Guessing" the enabler at every turn.

Hand coded error handling. Duplication of many features that are automatically provided by the 4GL product. Symptomatic of a 3GL method. The tool is "not trusted" or "not investigated" so everything is duplicated in long coding streams. Another symptom is that the automatic feature is "never exactly right" so the feature is duplicated in long code to achieve minimal or totally negligible benefits. The perceived benefits are usually only apparent to the programmer and do not affect end users in any way.

• No rules in dictionary.

Self evident. Larger programs, harder to maintain.

• No prompting programs.

Poor knowledge of new techniques. Rejection of CUA. Failure to see benefits of consistency. Repeat of large tracts of "search select" logic in every program.

• Fights natural flow of new technology at every turn.

Example of "ego" design. This is a very real problem. The designer sees the application as an expression of his/her own creativity. Rather than follow a corporate or global standard for dialogue design (like CUA), they evolve

their own design and then generate many lines of code to fight the natural CUA flow built into LANSA. Every application developed is then different to every other application. Also a symptom of resistance to change: "This is always the way it has been done and this is the way it will always be done". Sometimes this problem is not actually caused by the designer, but rather forced on him/her by the end users of the application.

• Much time wasted in screen panel layouts.

Typically the entire program is functionally complete in just one day, and then the programmer spends five more days trying to make the screen panels do things that are outside his/her capabilities. Real (as opposed to perceived) benefits to the end user are small or non-existent.

• Overloading of development environment.

Large programs, too many developers, deadline panic, poor work practices and unrealistic expectations usually combine to produce this type of development environment.

11.26 Large Numbers - Approach with Caution

There are many examples of very complex transactions available on the IBM i. Some are even shipped with the operating system.

For example:

- Signing on at a workstation.
- Signing off from the system.
- Starting a source edit session via STRSEU.
- Exiting and updating an edit session via STRSEU.
- Using the WRKACTJOB command.

These transactions exist and are used everyday, and they usually do not badly affect overall system operations.

The main reason that they do not affect system operations is that they all have a fairly low FREQUENCY OF USE. This means that at any given instant there is a FAIRLY LOW PROBABILITY that anyone is ACTUALLY USING the transaction.

However, this is not always the case.

Some sites have suffered from the "sign-off syndrome" at around 5pm when 150 - 200 users all attempt to sign off from the system in a 5 minute time span. In these situations the users who are not signing off have been known to suffer seriously degraded response times.

The most significant thing to observe here is that if you are dealing with any application that has a REASONABLE or HIGH frequency of use then you need to be very, very careful that it does not overload the machine.

Sometimes some simple mathematics will show up an "impossible" application. Unfortunately the "impossibility" of the application is sometimes not discovered until after it has been created and put into production.

Imagine a high volume order entry transaction that typically does 100 database accesses to validate and store order transactions.

The 100 database accesses is not an unreasonably high figure given the functionality of some order entry systems.

It is to be used continuously by 200 users who typically take 20 seconds to key in a transaction received over the phone.

This application has an extremely high probability of use.

Under peak load, every 20 seconds each user will be "requesting" 100 database accesses. That averages to 5 per second.

No multiply that by 200 users and you have a "requirement" for 1000 database accesses per second.

It should also be noted that these 1000 database accesses are "logical" accesses. In fact what is often counted as 1 "logical" access via a keyed access path results in many more "physical" disk drive I/Os as database file indexes are traversed and/or updated, etc. (e.g.: How many "physical" I/Os are required to do one "logical" write to a file that has 20 logical views - the answer is almost impossible to predict on a busy machine, but it will probably be a LOT more than 20).

Add to that normal machine virtual memory paging, work and job management for 200 users, etc, etc, and the 1000 "logical" accesses per second is probably something more like an overall application requirement for 5,000 to 10,000 actual disk accesses PER SECOND for this order entry system under peak load. (Note: the values 5,000 to 10,000 are used to illustrate a point - the actual figures on a busy machine would be virtually impossible to predict because of the hundreds of factors than can affect them - but they would certainly be much, much larger than a simple projection of the "logical" I/O counts).

The overall problem here is that the transaction is far too complex for its probability of use.

The main thing to note about avoiding a situation like this is that the FREQUENCY OF USE of a transaction, or even a whole system, is the most significant constraint to the level of functionality that it can provide.

If the FREQUENCY OF USE is moderate or high extreme caution is required to ensure that transactions are not "over" functioned.

If the high level of functionality is a business requirement, then a larger computer may be part of the cost of providing the new system.

Similar thought should be given to batch transaction processing where very LARGE VOLUMES of information are usually processed.

For instance a very complex batch transaction may be developed and tested on data sets of 10,000 records. The average elapsed time of the run is 30 minutes.

This is not a problem and the batch transaction goes into production. After a period of time in production it is processing runs of 1,000,000 records and producing extremely long elapsed run times.

The reason is again shown by simple mathematics. The production runs involve

a 100 times larger data set than the test runs. Multiple 30 minutes by 100 and you have 50 hours.

While this appears a very simple problem, and very easy to anticipate, you would be surprised at how often the simple equation $30 \text{mins} \ge 100 = 50$ hours is overlooked during application design and development.

Another situation that is overlooked sometimes is the effect of a program change. Imagine a report that is produced by sequentially reading and summarizing 1,000,000 records.

It has a run time of 2 hours.

The user wants another piece of information on the report. This requires an access to another file. Result: the run time will increase to 4, or possibly even 5 or 6 hours.

The purpose of this section and these examples is simply to demonstrate two points:

- If you are dealing with complex and/or heavily used online transactions, or with complex batch transactions processing many records, then caution is required.
- The amount of functionality that an application can provide must be limited by its probability of use, the volume of data that it processes, and most significantly, by the power of the computer that it is to be used on.

11.27 Estimating Application Performance by Running an Application

A common problem when developing applications for the IBM i is trying to estimate how they will actually execute on a "busy" or "loaded" production system.

If you test a function and "guess" that it seems to run okay, then you run the following risks:

• You may have run it many times before you do the test. This may mean that parts of the program or the other objects it accesses are still resident in the computer's main memory, thus reducing the time it takes for the program to be accessed.

For example, take the following simple program which does no initial processing, just puts up a screen panel:

REQUEST FIELDS(#PRODNO)

You may test this program and find that the panel appears in 1 second on average. However, you may find that if you sign onto the computer at the start of the day and then attempt to access the program initially it takes 4 seconds.

If you then repeatedly access the program from the menu, you find the access time gets less and less. The faster you access the program and return to the menu and (re)access the program the lower the access time seems to get. In fact you might find that the access time drops down to an average of 1/4 a second or less.

You might also notice that if you stop accessing the program for 5 minutes, and have a coffee break, your next initial access climbs back to 3 seconds or so.

How can this be, when you know that the program is doing exactly the same amount of work every time it is accessed (assuming that it is not an *HEAVYUSAGE program) ?

The reason is simple. When you access and re-access the program very quickly, the program and the objects that it accesses tend to remain in the computer's memory. However, if it is the very first access they are not in memory. Likewise, if you take a long break between interactions, it is likely that the objects are "rolled out" of memory back onto disk. If you wish to reaccess them, they must be "rolled in" to memory again, causing you to wait while this is done.

This phenomena is not new, and applies to many "virtual memory" operating systems such as the IBM i, but you should be aware of its existence when you are testing your applications.

The faster you work, the quicker the system will respond. The slower you work, the slower the system will respond. This effect is very pronounced in systems that have very long "think times" such as complex database inquiries, and least pronounced in system that have very short "think times" such as high speed data entry systems.

You should allow for this in any testing. If your system has a long "think time", you should emulate this in testing, or you may be using the operating system in a way that does not truly reflect the way that users of your system will.

- You may be on a totally different type of machine or using a very different type of environment. If you do your test on an IBM i 270 at 9.00pm at night, when everybody else has gone home, and the actual target machine is a very busy 170, then the results you obtain will be totally misleading.
- The system may not be busy enough. When the IBM i begins to become loaded up with work, its performance characteristics change.

The transaction throughput curves for most IBM i rise slowly until a certain load is reached. At this point the curve rises almost vertically, so that almost no more work is being done, but the response times degrade very rapidly.

If you are using a system that is operating near to this "knee of the curve" you will find that a relatively small increase in the machine load will dramatically degrade the response time of all online users and batch jobs.

Hopefully all of these points combined are enough to convince you that attempting to estimate how an application will perform in a production system is not a simple task and it should not be just doing by some simple testing and then concluding that "it seems to run okay".

11.28 Estimating Application Performance by Calculating an ERUF

Given that it is very hard to really test your application's response time on a computer, without using the actual target machine with the real end users signed on and using the system (which is a rather late stage to do such a test), you might find that it is better to apply a "paper estimation" to your critical transactions.

You could base such a test on the following calculation tables, which produce an "ERUF" (Estimated Resource Utilization Factor) for all, or parts of, individual transactions.

An ERUF indicates the type of relative response time you could expect, given the amount work that your function does.

Before attempting to use these ERUF calculation tables, you should be aware of the following:

11.28.1 The Recommended ERUF Ranges are for BUSY Production Machines

11.28.2 ERUF Values are NOT Response Times

11.28.3 Associating ERUF Values With Response Times is a MAJOR Mistake

11.28.4 ERUF Values for Batch Transactions do NOT indicate Elapsed Time

11.28.5 Realistic Estimates are Important

11.28.6 Recommended ERUF Ranges

11.28.7 A High ERUF Value Does Not Mean That Your Transaction Is Invalid 11.28.8 ERUF Calculation Table

11.28.1 The Recommended ERUF Ranges are for BUSY Production Machines

This may occasionally upset your designs for a specific function.

For instance, an ERUF "entry logic" value in the range 1 to 5 is recommended. You may have just designed a complex online inquiry that opens 17 files and does 350 databases accesses to present a summary panel. You calculate an ERUF value of 43 for your program, which is far into the "very complex" transaction range.

The very high ERUF value you have calculated does not mean that your function cannot work and should not be used, but it is a clear warning to you that on a busy production machine you must not expect it to deliver a very fast response time, and also that it is a very heavy system resource user.

You may even feel that the ERUF is misleading, because the tests you have done in the evening on your B60 indicate a response time of 6 seconds. However, the ERUF is clearly warning you that such a result is not typical, and a more likely outcome on a busy machine is a much longer response time (probably 20 - 40 seconds).

11.28.2 ERUF Values are NOT Response Times

The ERUF value that you calculate is not a response time. It is just a number that allows you to compare your applications with recommendations made in this manual, and with applications that you have already created.

It is in an indication that what you want to do is "fair and reasonable" in terms of resource utilization, and allows you to relate the resource requirements to other functions that you have created.

For instance, if you have many transactions with ERUF=4 values and they work well, then it is very likely that a new transaction with ERUF=5 will work almost as well, but a transaction with ERUF=12 will take about 3 times as long to complete.

11.28.3 Associating ERUF Values With Response Times is a MAJOR Mistake

It may be tempting to try to associate ERUF values with response times to estimate an ERUF "tolerance range" for your specific site.

Attempting such a process is very dangerous and can lead you to make serious mistakes that will cause many problems later.

For example, you may create several ERUF=4 functions and test them on a brand new (and very under utilized B60) to get an average response time of 1 second. Since you allow responses of up 2.5 seconds, you therefore conclude your "tolerance range" is from 1 to 10 ERUF units for any transaction.

However, the reality is that when the IBM i is heavily loaded with 50 online users and 7 batch jobs, the ratio changes and your ERUF=4 transactions are now averaging 3 seconds response.

Where does this leave your ERUF=10 transactions ? Probably with a response time of around 30 seconds.

11.28.4 ERUF Values for Batch Transactions do NOT indicate Elapsed Time

The association of an ERUF value for a batch program with a batch job with an elapsed throughput time is very dangerous as well. For instance, you may do tests that conclude one ERUF unit is 1 second elapsed in batch jobs. However, the next day, higher priority online jobs are using more CPU cycles, so the ERUF ratio may stretch to 2 or 3 seconds, thus making your batch job take 2 or 3 times longer to complete (in elapsed time).

The ERUF value for batch jobs can really only be used to compare the complexity of a new batch job with the complexity of any existing batch jobs that you have.

11.28.5 Realistic Estimates are Important

When you are calculating an ERUF value, some sections of code will require you to estimate. For example, if the entry logic in a transaction had the following RDML commands:

```
SELECT FIELDS(.....) FROM_FILE(.....) WITH_KEY(....)
ADD_ENTRY TO_LIST(.....)
ENDSELECT
```

then you will have to realistically estimate how many records the select loop will find and add to the list. If you estimate 20, then this will add 2.0 to your entry logic ERUF value, which will probably keep the transaction in the simple range.

However, if the reality is that 500 will be added, then this will add 50.0 to your ERUF value, pushing the transaction well into the very complex range.

The importance of this effect is sometimes overlooked when creating applications, but the reality is that for this particular piece of RDML code, the second version does 25 times more work than the first - even though it does not look to be 25 times more "complex".

11.28.6 Recommended ERUF Ranges

The following calculation tables produce an ERUF value. The classification of the calculated value for online transactions should be made like this:

ERUF From Value	ERUF To Value	Description Of Function	Comments
1	5	Simple transaction	Should be okay.
6	10	Medium transaction	Take some care with usage.
11	20	Complex transaction	Watch usage very carefully.
	> 20	Very Complex transaction	Avoid if possible.

For batch transactions, the value can be very large indeed, and it simply indicates to you the complexity of the batch transaction relative to other batch transactions you have created.

11.28.7 A High ERUF Value Does Not Mean That Your Transaction Is Invalid

If you design a transaction and calculate an ERUF value of 35, then it is obviously a very complex transaction, but what does this high value of 35 indicate to you?

- That if this transaction is part of a heavily and frequently used application, you should consider simplifying it, because it is too complex for heavy and frequent use on most IBM i.
- That if this application is not used frequently it is probably okay but it could become a "bottleneck" if its usage rates change. Its usage rate can change because it used more often by the same group of users, or because new groups of users start to use it.

11.28.8 ERUF Calculation Table

Use this table to calculate an ERUF value for all or part of a transaction. Typical parts of a critical transaction that are checked by using this table include:

- Resource needed to invoke a function and display first panel.
- Resource needed to progress from a panel to next panel.
- Resource needed to close down function and return to caller.
- Resource needed to process entire batch job.

Resource Usage Item	Calculation	Est. ERUF
Entry to small function (<200 lines)	Add 1	
Entry to medium function (<500 lines)	Add 2	
Entry to complex function (<900 lines)	Add 3	
Entry to very complex function (>900 lines)	Add 4	
Does not use OPTIONS(*NOMESSAGES). For entry and exit logic only.	Add 0.25	
Does not use OPTIONS(*DEFERWRITE). For any code using screen panels.	Add 0.25	
Display panel presented on remote device.	Add 1	
Exit logic only. If any external built-ins are used, add one only shutdown value of	Add 0.25	
Open off shared files already opened by another higher level function.	Divide total by 4 and add the result	
Close off shared files that are still open in another higher level function. Usually only applies to exit logic.	Divide total by 8 and add the result	
Open of files that are not already open	Divide total by 2 and add the result	
Close of files not opened by higher caller Usually only applies to exit logic.	Divide total by 4 and add the result	

CALLs to other functions. Add the total ERUFs of all called functions.	Add all ERUFs
Database I/Os performed	Divide total by 10 and add the result
Group 1 RDML commands / built-ins executed (count loops repetitively)	Divide total by 100 and add the result
Group 2 RDML commands / built-ins executed (count loops repetitively)	Divide total by 50 and add the result
Group 3 RDML commands / built-ins executed (count loops repetitively)	Divide total by 20 and add the result
Group 4 RDML commands / built-ins executed (count loops repetitively)	Divide total by 10 and add the result
Total ERUF value	n/a
For *HEAVYUSAGE re-entry, subtract file opens	n/a
For re-execution of code that has to open files the first time that it is executed, subtract file opens	n/a

11.29 Breakdown of RDML Commands into Resource Groups

Group notations: /w = when used with working

/b = when used with a browse list

n/a = not applicable to this command

Command Group

Command	Resource Group	Command	Resource Group	Command	Resoi Grou
ABORT	1	ENDCHECK	1	MENU	1
ADD_ENTRY	1/w 2/b	ENDIF	1	MESSAGE	3
BEGIN_LOOP	1	ENDPRINT	3	ON_ERROR	1
BEGINCHECK	.1	ENDROUTINE	1	OPEN	3
BROWSE	4	ENDSELECT	1	OTHERWISE	1
CALL	n/a	ENDUNTIL	1	OVERRIDE	n/a
CALLCHECK	n/a	ENDWHILE	1	POINT	1
CASE	1	EXCHANGE	2	POP_UP	4
CHANGE	1	EXEC_CPF	n/a	PRINT	3
CHECK_FOR	3	EXEC_OS400	n/a	RANGECHECK	1
CLOSE	3	EXECUTE	1	RENAME	n/a
CLR_LIST	1/w 2/b	EXIT	1	REQUEST	4
COMMIT	3	FETCH	3	RETURN	1
CONDCHECK	1	FILECHECK	3	ROLLBACK	3
DATECHECK	2	FUNCTION	n/a	SELECT	3
DEF_ARRAY	n/a	GET_ENTRY	1/w 2/b	SELECTLIST	1/w 2.
DEF_BREAK	n/a	GOTO	1	SET_ERROR	2
DEF_COND	n/a	GROUP_BY	n/a	SET_MODE	1
DEF_FOOT	n/a	IF	1	SKIP	2
------------	---------	------------	---------	------------	--------
DEF_HEAD	n/a	IF_ERROR	1	SORT_LIST	4
DEF_LINE	n/a	IF_KEY	1	SPACE	2
DEF_LIST	n/a	IF_MODE	1	SUBMIT	4
DEF_REPORT	n/a	IF_NULL	1	SUBROUTINE	1
DEFINE	n/a	IF_STATUS	1	SUBSTRING	3
DELETE	3	INSERT	3	TRANSFER	1
DISPLAY	4	INZ_LIST	1/w 2/b	UPD_ENTRY	1/w 3.
DLT_LIST	1/w 2/b	KEEP_AVG	1	UPDATE	3
DOUNTIL	1	KEEP_COUNT	1	UPRINT	4
DOWHILE	1	KEEP_MAX	1	USE	n/a
ELSE	1	KEEP_MIN	1	VALUECHECK	1
END_LOOP	1	KEEP_TOTAL	1	WHEN	1
ENDCASE	1	LOC_ENTRY	2		

11.30 Breakdown of Commonly Used Built-in Functions into Resource Groups

Built-In Function	External	Group
ACCESS_FILE	Y	4 + I/O
BCONCAT	Y	2
CENTRE	Y	2
CHECK_AUTHORITY	Y	2
CHECKNUMERIC	Y	2
CHECKSTRING	Y	2
CLR_MESSAGES	Y	2
CONCAT	Y	2
CONVERTDATE	Y	2
DATEDIFFERENCE	Y	2
EXCHANGE_ALPHA_VAR	Ν	1
EXCHANGE_NUMERIC_VAR	Ν	1
EXPONENTIAL	Y	3
FILLSTRING	Ν	2
FINDDATE	Y	2
GET_CHAR_AREA	Y	3
GET_MESSAGE	Y	3
GET_MESSAGE_DESC	Y	3
GET_NUM_AREA	Y	3
ISSUEINQUIRY	Y	3
ISSUEMESSAGE	Y	3
LEFT	Y	2

PUT_CHAR_AREA	Y	2
PUT_NUM_AREA	Y	2
RESET_@@UPID	Y	4 + all I/Os
REVERSE	Y	2
RIGHT	Y	2
ROUND	Ν	2
SCANSTRING	Y	3
SET_ACTION_BAR	Ν	2
SQUARE_ROOT	Ν	1
TCONCAT	Y	2
UPPERCASE	Y	3

12. IBM i SAA/CUA Implementation

12.1 LANSA SAA/CUA Overview
12.2 SAA/CUA Partitions
12.3 SAA/CUA Panel Elements
12.4 SAA/CUA Panel Element Attributes
12.5 SAA/CUA Function Key Assignments
12.6 SAA/CUA Process Menus
12.7 SAA/CUA Help Text

12.8 SAA/CUA User Defined Panels

12.9 SAA/CUA Function Key Use

12.1 LANSA SAA/CUA Overview

LANSA supports the concepts of SAA (Systems Application Architecture) and the panel design and user interaction concepts prescribed by the CUA (Common User Access) guidelines.

This section is about how the SAA/CUA guidelines and standards are implemented within LANSA, it is not about what the actual SAA/CUA guidelines and standards are. For this information refer to the appropriate IBM supplied manuals, such as *Systems Application Architecture: Common User Access Basic Interface Design Guide (SC26-4583)*.

Before attempting to implement (or not to implement) applications in LANSA that use the prescribed SAA/CUA guidelines consider the following points:

• SAA/CUA is an IBM strategic direction. As such the LANSA product has a strong commitment to the concept. While all existing LANSA features will always be supported for non-SAA/CUA applications, new features that become available may only be accessible from SAA/CUA compliant applications.

Note that this does not mean you have to convert all your existing applications to be SAA/CUA compliant (which usually only means recompiling them anyway), but you will have to eventually set up SAA/CUA parameters for all LANSA partitions being used.

While an SAA/CUA compliant partition will readily run non-SAA/CUA RDML applications, a non-SAA/CUA partition cannot run SAA/CUA compliant RDML applications.

• The LANSA implementation of SAA/CUA described by this appendix adheres to the standards and guidelines layed down in the IBM manual *System Application Architecture: Common User Access Basic Interface Design Guide (SC26-4583).*

It does not necessarily adhere to the recommendations made in the IBM i document "*Defining AS/400 Compatible Displays using Data Description Specifications Newsletter*" (*GC21-8163*) and other IBM i publications.

Some of the reasons for this are:

• LANSA supports action bars, pull-downs and pop up windows.

- LANSA supports number or cursor selection.
- In some areas the recommendations do not conform to the SAA/CUA guidelines. For instance, messages should appear above the function key area, not below it.
- The use of SAA/CUA standards requires a firm commitment at the corporate level.
- The full use of SAA/CUA guidelines may meet resistance from some EDP practitioners, in particular those that have worked on System/38s for a long time. They may feel that in many respects the SAA/CUA guidelines are too restrictive.
- The use of some of the features available within the SAA/CUA guidelines such as action bars, pull-downs, pop up windows and the prompt key may significantly change the way application systems are designed. Different architectural methods are generally used for SAA/CUA application systems that use these facilities.

12.2 SAA/CUA Partitions

SAA/CUA is implemented by LANSA at the partition level. The following points about this feature should be noted before attempting to create an SAA/CUA partition or change an existing one to use SAA/CUA guidelines:

- A non-SAA/CUA partition cannot contain SAA/CUA processes.
- An SAA/CUA partition can contain non-SAA/CUA processes.

12.3 SAA/CUA Panel Elements

When a user executes a LANSA process or function he or she may interact with a series of panels. A panel is what appears on the screen/workstation and is viewed by the end user.

Every single thing that appears on a LANSA panel **must** fall into one of the following categories:

Panel Element Category	Description Of Panel Element Category
АВСН	Action bar and pull-down choices
PBPT	Panel title
РВРІ	Panel identifier
PBIN	Instructions to user
PBFP	Field prompt / label / description details
PBBR	Brackets
РВСМ	Field column headings
PBGH	Group headings
PBNT	Normal text
PBET	Emphasized text
PBEN	Input capable field (normal)
PBEE	Input capable field (emphasized)
РВСН	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

When a new LANSA partition is being defined to conform to SAA/CUA standards (or an existing one is being modified), you are asked to define 3 different things for each of these panel element categories.

These are:

- What color elements in this category should be displayed on color screens?
- Any special attributes that apply to panel elements in the category when it is used on a color screen?
- Any special attributes that apply to panel elements in the category when it is used on a monochromatic screen?

For example, the category PBEE (emphasized entry field) is usually set up with color white, no special color screen attributes, and with special attribute HI (high intensity) on monochromatic screens.

This means that on every panel presented by LANSA within the partition, any emphasized entry field will appear in white on color screens and in high intensity on monochromatic screens. (Note that this does not mean high intensity white on color screens).

These panel body categories can actually be associated with fields as input or output attributes within the LANSA data dictionary. Usually only four of these are ever used in this way:

Panel Element Category Description Of Panel Element Category

PBEN	Input capable field (normal)
PBEE	Input capable field (emphasized)
PBCN	Protected field (normal)
PBCE	Protected field (emphasized)

In an SAA/CUA partition, a vital or key field, such as "Customer Number" might be defined in the LANSA dictionary like this:

Input attributes : PBEE FE RB Output attributes : PBCE Whereas a normal field, such as "Customer zip code" might be defined in the LANSA dictionary like this:

Input attributes : PBEN RZ

Output attributes : PBCN

Note how additional input or output attributes can also be included into this data dictionary list. Typically, these include things like FE (field exit required), RB (right adjust and blank fill), etc, etc.

The use of panel element categories in this way reflects the whole philosophy of using SAA/CUA.

The analyst defining the fields in the dictionary can decide whether the field is to be emphasized or normal.

How an emphasized field or a normal field appears on a panel has already been defined as a corporate standard. The analyst need not worry about this.

Even if the corporate standard is reverse video, blinking, in color pink, it should be followed. This is what CUA (common user access) is all about.

12.4 SAA/CUA Panel Element Attributes

The following color and attribute table describes the default colors and attributes adopted by LANSA itself, and for applications created within SAA/CUA partitions.

You are not forced to follow these default values.

Panel Element Category	Color Additional Color Screen Attributes	Additional MonoScreen Attributes
ABCH Action bar and pull-down	White	High Intensity
PBPT Panel title	Blue	
PBPI Panel identifier	Blue	
PBIN Instructions to user	Green	
PBFP Field label or description	Green	
PBBR Brackets	Green	
PBCM Field column headings	Blue	High Intensity
PBGH Group headings	Blue	High Intensity
PBNT Normal text	Green	
PBET Emphasized text	White	High Intensity
PBEN Input capable field (normal)	Green	
PBEE Input capable field (emph)	White	High Intensity
PBCH Choices shown on menu	Green	
PBSC Choice last	White	High Intensity

selected on menu

PBUC Choices not available	Blue	
PBCN Protected field (normal)	Green	
PBCE Protected field (emphasized)	White	High Intensity
PBSI Scrolling information	White	High Intensity
PBSL Separator line	Blue	
PBWB Pop-up window border	Blue	
FKCH Function key information	Blue	

12.5 SAA/CUA Function Key Assignments

When a new LANSA partition is being defined to conform to SAA/CUA standards (or an existing one is being modified), you are asked to define what standard function key assignments are to be used and what the Short Form description of the function key is to appear on all generated screens.

The table below indicates the standard function key assignments used by the LANSA implementation under SAA/CUA, the suggested Short Form descriptions and the suggested Function Key that should be assigned.

Like the panel attributes facility, you are not forced to follow the recommended values and can change them to your site standards if desired.

Notes	Description Of Function Key	Suggested Short Form	Suggested Function Key
Re	Cancel current action	Cancel	F12
Re	Enter any entry fields	Enter	Enter
Re	Request help for the current function	Help	F1
Ор	Display choices from a list of entries	Prompt	F4
Op *	Restore or refresh panel	Refresh	F5
***	LANSA defined messages key	Msgs	F14
Op *	Scroll panel backward	Bkwd	F7
Op *	Allow entry of a command	Command	F9
Op *	Display cycle of function key area forms	Keys	F13
Re	Exit to the highest level	Exit	F3
Re	Exit to the next level above	Exit	F3
Op *	Move cursor to first field on the screen	Home	Home
Op *	Scroll panel forward	Forward	F8

Op *	Display table of contents for help	Contents	F23
Ор	Provide information about entire panel	Ex Help	F2
Ор	Provide index of help information	Index	F11
Op *	Provide names and functions of keys	Keys help	F9
Op *	Scroll panel left	Left	F19
Op *	Re-display last command that was issued	Retrieve	F9
Op *	Scroll panel right	Right	F20
Op *	Move cursor backward	Switchbkwd	F18
Op *	Move cursor forward	Switch fwd	F6
Ор	Move cursor to action bar	Actions	F10
***	LANSA defined add key	Add/Create	F6
***	LANSA defined change key	Change	F21
***	LANSA defined delete key	Delete	F22

Notes about this table:

Re = denotes an SAA/CUA "reserved" function key. In such cases the function key cannot be reassigned to other functions, even if the specified functions don't apply to a particular panel.

Op = denotes a "non-reserved" SAA/CUA function key. In such cases the function key may be reassigned to other functions, but only if the application does not support the functions prescribed for the function key by this table. * = denotes an SAA/CUA function key assignment that is prescribed for complete SAA/CUA compatability, but is not currently implemented directly within LANSA. The use of such keys is controlled by the USER_KEYS parameter of DISPLAY, REQUEST or POP_UP commands. It is the responsibility of the user to implement such key assignments in RDML programs as per the SAA/CUA guidelines described by the table.

*** = denotes a non-SAA/CUA proscribed function key that is required in this table because it is implemented by LANSA.

12.6 SAA/CUA Process Menus

A process menu that uses SAA/CUA as implemented by LANSA will appear like one of these examples:

When "Number" Style Menus Are Specified for the Partition

"Number" style menus are the SAA/CUA recommended style.

SAATEST	Example SAA/CUA test process
Enter number of	function required or place cursor on same line
 1. Display we 2. Work with e 3. Work with e 4. Work with e 5. Display pro 6. Return to m 	rkstation messages customers orders communication links cess or function HELP text ain system menu
F1=Heln F3=F	xit F12=Cancel F14=Messages

When "Cursor" Style Menus Are Specified for the Partition

SAATEST Example SAA/CUA test process

Place cursor on same line as function required and press enter

Display workstation messages Work with customers Work with orders Work with communication links Display process or function HELP text Return to main system menu

Some points to notice about these menus are:

- The **style of menu** (number selection or cursor selection) is defined at the partition level. All menus in the partition are one style or the other. Intermixing of styles is not permitted.
- The **panel identifier** (SAATEST) is the name of the LANSA process being used.
- The **panel title** (Example SAA/CUA test process) is the description of the LANSA process being used. It is automatically centred, and may be optionally automatically converted to uppercase (uppercase conversion requirement is defined at the partition level).
- The **time and date** may be optionally displayed (defined at the partition level). The default in most partitions is to not display the time and date.
- The **instruction line** (line 3) is up to 76 characters long. It is site definable (at the partition level).
- In compiled processes the **menu number entry field** may be 1 or 2 characters long (dependent upon the number of entries on the menu). When executed interpretively, the input field is always 2 characters long.
- Additionally, the **menu number entry field** may be defined to use the AUTO(RA) (automatic record advance) feature. This prevents users having to press the enter key after entering the number of the function they wish to use. This use of this facility can be switched on or off at the partition level.
- **No spacing lines occur between any menu entries** (which differs from the implementation of menus for non-SAA/CUA processes).
- The **display HELP text and return to previous higher level menu** options always appear as the last 2 options on any menu.

- **Messages** always on line 22 in interpretive processes. In compiled processes, messages may appear on line 22 or line 24. The position is defined at the partition level and applies to all compiled process menus.
- **Function keys** appear on lines 23 and 24 in interpretive processes. In compiled processes function keys may appear on lines 23/24 (with messages on line 22) or on lines 22/23 (with messages on line 24). The positions are defined at the partition level and apply to all compiled process menus.
- The following **color and attribute** table applies to interpretive process menus. It may not apply to compiled process menus if the SAA/CUA recommended standard color and attribute defaults for panel element categories have been modified:

Panel Component	Default Color	Other Attributes on Color Device	Attributes On Mono Device
Identifier	Blue	None	None
Title	Blue	None	None
Date and time	Blue	None	None
Instructions	Green	None	None
Menu choice	Green	None	None
Selected choice	White	None	High intensity
Messages	White	None	High intensity
Function keys	Blue	None	None

- When using a **bi-directional language**, within a multilingual partition, details on the menu are "reversed" as if they were reflected in a mirror.
- When using a **DBCS language**, within a multilingual partition, all textual details can be made to appear as ideographic characters.

12.7 SAA/CUA Help Text

When a user presses either the HELP key or the function key assigned to SAA/CUA help facility (usually F1) a window will appear that overlays the lower portion of the current panel as in this example:



Some points to notice about this HELP menu are:

• The **panel identifier** (IIIIIIIIII) is the name of the LANSA process, function

or field that was used when the HELP facility was invoked. In the case of a process the process name is shown as the panel identifier. In the case of a function the function name is shown in the format ffffffHLP. Any imbedded blanks in the name are filled with "_" (underscore) characters. In the case of a field the field name is shown.

- The **panel title** (TTTTT.....TTTTT) is the description of the LANSA process, function or field being used at the time the HELP facility was invoked. It is automatically centred, and may be optionally automatically converted to uppercase (uppercase conversion requirement is defined at the partition level).
- The **time and date** is not shown in any HELP window. The overlayed process or function panel will show the date and time if it is enabled within the partition.
- **Messages** always on line 24. The IBM i operating system does not allow a message subfile to be successfully placed within a window at the current time.
- **Function keys** appear on lines 21 and 22. They relate to the HELP facility window, not to the process or function that was in use at the time the HELP facility was invoked. Where only one line of function keys apply, they will be displayed on line 22. This creates a visual separation between the HELP text and the function key area.
- The following color and attribute table applies to HELP text windows. Since HELP text windows are "interpretive", these colors and attributes will always apply, no matter how color and attribute defaults for panel elements have been modified within the current partition:

Panel Component	Default Color	Other Attributes on Color Device	Attributes On Mono Device
Identifier	Blue	None	None
Title	Blue	None	None
Text	Green	None	None
Window Blue border		None	None
Messages	White	None	High intensity
Function	Blue	None	None

keys

- When using a **bi-directional language**, within a multilingual partition, help details are "reversed" as if they were reflected in a mirror.
- When using a **DBCS language**, within a multilingual partition, all textual details can be made to appear as ideographic characters.
- **Function key 2** is for "Extended HELP" and is enabled only when reviewing the HELP text associated with a FIELD. Using this function key causes any extended HELP text to be displayed. Typically, extended HELP text is the HELP text associated with the FUNCTION that is currently in use.

For example, a user may be running a customer maintenance function, and uses HELP or F1 when the cursor is positioned in the customer number field. Initially the HELP text associated with the field customer number will be displayed. If the user then uses F2, the help text associated with function (maintain customers) will be displayed.

• If **Function key 11** (or the partition defined equivalent) is used to request that the HELP index be displayed, a new window will overlay the current window like one of the following examples:

When "Number" Style Menus Are Specified for the Partition

SAATEST Example SAA/CUA test process

Enter number of function required or place cursor on same line

_ 1. Display workstation messages

- 2. Work with customers
- 3. Work with orders
- 4. Work with communication links

HELP_INDEX HELP Index | Enter number of desired topic or put cursor on same line | and press enter | More: + |



When "Cursor" Style Menus Are Specified for the Partition

SAATEST Example SAA/CUA test process 03/03/89 12:00:00						
Place cursor on same line as required function and press enter						
Display workstation messages Work with customers Work with orders						
Work with communication links						
HELP_INDEX HELP Index						
Place cursor on same line as desired topic and press enter						
More: +						
Process - xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
Function - xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
Function - xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
Field - xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
Field - xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						

12.8 SAA/CUA User Defined Panels

To understand the way using an SAA/CUA partition affects the layout and attributes of a user defined panel created by an RDML DISPLAY, REQUEST or POP_UP command, consider the following example:

MNTCUST14 Maintain customer details 03/03/89 12:00:00

Enter customer number or select customer from list.

Customer No . . .

	Customer		Zip	
Sel	Number	Customer name	Code	
•	999999	XXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXX
•	999999	XXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXX
•	999999	XXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXX
•	999999	XXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXX
•	999999	XXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXX
•	999999	XXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXX

F1=Help F3=Exit F12=Cancel F14=Messages

For this example the following SAA/CUA characteristics are worth noting:

- The **panel identifier** is derived from the RDML function name and the sequence number of the RDML program statement that caused it to be displayed. It uses the attributes defined for panel element category PBPI.
- The **panel title** is the description of the LANSA function. It is automatically centred, and may be optionally automatically converted to uppercase. It uses the attributes defined for panel element category PBPT.
- The **time and date** may be **optionally** displayed. If displayed, they use the attributes defined for panel element category PBCN.

- The **instruction line** (line 3) has been defined by the TEXT parameter on the associated RDML DISPLAY, REQUEST or POP_UP command. It uses the attributes defined for panel element category PBNT.
- **The data dictionary label "Customer No . . . "** used to identify the customer number field uses the attributes defined for panel element category PBFP.
- The actual input field **customer number** uses any SAA/CUA panel element category input attribute defined for it at the LANSA data dictionary level (or any specific overrides as may have been applied within this RDML program).

Since customer number is a "key" field it would be expected to have panel element category input attribute PBEE defined at the dictionary level.

If no SAA/CUA panel element category has been defined for the field at the dictionary or program level, panel element category PBEN would be automatically applied as an input attribute.

- All **column headings** use attributes as defined by panel element category PBCM.
- The **selection column** field would use attribute as defined by the data dictionary or the program. If none were defined, panel element category PBEN would automatically apply.
- The **customer number field** (in the browse list) would use SAA/CUA attributes as defined in the dictionary or program. If none were defined, panel element category PBCN would automatically apply.

The same rule applies to **all other fields** in the browse list.

- **Messages** may appear on line 22 or on line 24. The position is defined at the partition level and applies to all panels defined within the partition.
- **Function keys** may appear on lines 23/24 (with messages on line 22) or on lines 22/23 (with messages on line 24). The positions are defined at the partition level and apply to all panels defined within the partition.

They are displayed using the attributes defined for panel element category FKCH.

There are 2 full lines of function key area available. No key will ever "wrap"

onto the second line. It will appear fully in either the first or second line.

If too many keys are enabled to appear in the 2 lines provided, those that will not fit will not appear at all. No "roll" facility for the function key area is currently supported by LANSA.

- The **Help function key** is shown as F1=Help. Note that the actual HELP key on the workstation may also be used to invoke the HELP facilities.
- The **Exit function key** is by default a "high" exit, which causes a full exit from the entire application. If full SAA/CUA standards are used it will appear as F3=Exit.

If the specific DISPLAY, REQUEST or POP_UP commands in the RDML program indicates via the EXIT_KEY parameter that a "low" exit is required (ie: exit from this facility only), it would also be shown as F3=Exit.

• The **Cancel function key** is actually the function key controlled by the the MENU_KEY key parameter in RDML DISPLAY, REQUEST or POP_UP commands.

If full SAA/CUA standards are complied with it would appear as F12=Cancel. This is interpreted as meaning "cancel the current function and re-display the last menu that was used."

• When using a **bi-directional language**, within a multilingual partition, many details are "reversed" as if they were reflected in a mirror.

For instance labels and descriptions appear to the right of the field and column headings are aligned to the right on all fields (instead of just numeric fields).

• When using a **DBCS language**, within a multilingual partition, all textual details can be made to appear as ideographic characters.

12.9 SAA/CUA Function Key Use

In SAA/CUA partitions it is recommended that the EXIT_KEY and MENU_KEY parameters of the LANSA RDML commands DISPLAY, REQUEST and POP_UP be examined closely before attempting to design or implement a system.

In an SAA/CUA partition there are 2 different types of EXIT key enabled. The first is called a "high exit" and the second is called a "low exit". They are both usually assigned to function key 3.

LANSA defines the EXIT_KEY parameter as a "high exit" by default. When used, a high exit causes an exit from the entire application system to occur (unless specifically trapped by the calling routine(s)).

However, on any individual DISPLAY, REQUEST or POP_UP RDML command you can specify that a "low exit" is required, and that when it is used, control should be returned to the routine that **invoked** the routine containing the DISPLAY, REQUEST or POP_UP command.

The routine to which control is returned may be the calling RDML function, the program mainline, or even another subroutine.

LANSA also defines a standard function key called the MENU key (usually function key 12). It is often referred to in SAA/CUA partitions as the **CANCEL** key.

By default, use of the MENU/CANCEL function key causes the executing RDML function to end and the last menu that was displayed to be re-displayed.

However, on any individual DISPLAY, REQUEST or POP_UP RDML command you can specify the current function should be cancelled and that control should be returned to the routine that invoked the routine containing the DISPLAY, REQUEST or POP_UP command.

Again, the routine to which control is returned may be the calling RDML function, the program mainline, or even another subroutine.

As a general rule, in "mainline" programs (ie: those directly accessible from menus) use the following values for the EXIT_KEY and MENU_KEY parameters:

EXIT_KEY(*YES *EXIT *HIGH) or EXIT_KEY(*YES)

MENU_KEY(*YES *MENU) or MENU_KEY(*YES)

Similarly, in "subroutine" programs (ie: those that perform services for calling

programs / routines) the following values for the EXIT_KEY and MENU_KEY parameters are recommended:

EXIT_KEY(*NO) or EXIT_KEY(*YES *RETURN *LOW) MENU_KEY(*YES *RETURN)

While there is no real need to exactly follow these requirements, what is important is that an application standard for EXIT and MENU function keys is established before one single RDML program is written and is understood and adhered to by all programs and programmers.

13. Generated Code C and SQL/ODBC Considerations

- 13.1 Overview
- 13.2 Object Naming Under Windows
- 13.3 Process Name Truncation and Duplicate Object Names
- 13.4 Automatic Renaming for SQL/ODBC
- 13.5 Accessing the Generated C Code
- 13.6 Handling Relative Record Numbers
- 13.7 Renumbering Relative Record Numbers
- 13.8 U_TEXT Strings and .UTX Files
- 13.9 The SLLLLOOO & DLLLLOOO.DAT Tables
- 13.10 Things That May Be Different
- 13.11 ASCII Versus EBCDIC Data Collating Sequences
- 13.12 Commitment Control
- 13.13 SQL/ODBC Grammar: Keyword Conflicts
- 13.14 General ODBC Driver Limitations

13.1 Overview

This section outlines important details regarding the C and SQL/ODBC implementation used by Visual LANSA. You should be familiar with this section of the guide before designing and building your application database or new applications.

If you have never used SQL or ODBC before, i.e. you have only used LANSA/AD and are translating your application using Visual LANSA, you should take some time to learn some SQL/ODBC fundamentals.

Detailed knowledge of SQL/ODBC is not a requirement. LANSA insulates the developer from the specific platform requirements; however, knowledge of the target platform will assist you in developing your applications.

For example, SQL/ODBC has a number of keywords which have special meaning to SQL/ODBC. These keywords (e.g. ADD, COMMENT, INSERT) are reserved and cannot be used as field names in an SQL/ODBC table. Again, LANSA will insulate you from this using automatic renaming facilities; however, when designing your database or when accessing the SQL/ODBC tables directly, this type of knowledge will be helpful.

A partition identifier containing the special characters "\$@" (in this sequence) will NOT be supported. This is the result of restrictions when using a "make" file to compile and link Visual LANSA objects.

For example a partition identifier of "L\$@" is NOT supported, but "L@\$", "LX@" and "L\$X" are supported.

Use of an "@" sign anywhere in a partition name under Windows environments is not supported and you should avoid using partitions that have an "@" sign in their name when working with Windows.

Portability
ConsiderationsNote: A recommended portability standard for all
partitions identifiers is to use letters of the English
alphabet (A - Z) and numbers (0 - 9) only.

13.2 Object Naming Under Windows

One of the well known differences between the IBM i and other operating environments is the length of object names.

These differences are described in detail in:

- *Portability Considerations* in the *Data Dictionary*
- Portability Considerations with the Database
- Portability Considerations for Processes and Functions.

Historically, most non-IBM i environments only allow 8 character names while the IBM i allows 10 character names.

Visual LANSA has been designed to automatically rename objects for you when an IBM i application definition is migrated. You should be aware of the following:

- The naming standards and length limitations are historically imposed by the non-IBM i operating environments.
- Visual LANSA automatically renames files, processes and functions to conform to these rules
- All dictionary and RDML references remain unchanged, i.e. you may still reference a 10 character object name in your RDML
- New systems being developed should take note of these standards and avoid using names outside these standards.

If you are planning to access the SQL tables directly or work with the generated C code, you should be aware of the renaming requirements.

Portability
ConsiderationsA recommended portability standard for all new
application names is to use letters of the English alphabet
(A>Z) and Arabic numbers 0 > 9 only in names. Limit all
object names to 8 characters or less in length.

13.3 Process Name Truncation and Duplicate Object Names

Process names are truncated on the left as they are converted to equivalent non-IBM i object names.

If you are using a naming standard like "ooooPROCnn" (where "oooo" is the object name) then you might have several processes named like this:

CUSTPROC01 (CUST = Customer)

OUSTPROC01 (OUST = Overseas Customer)

DUSTPROC01 (DUST = Domestic Customer)

In this case, the left truncated name for all of these processes would be defaulted by Visual LANSA to be STPROC01, which is a clear conflict.

In situations where the default left truncation algorithm causes duplicated names, you can solve this problem by using the Process Alias file x_palias.dat.

x_palias.dat is a normal source file that may optionally exist in the <drive/root>\x_lansa directory. It is edited by any normal source editor such as EDIT or EPM, can have at most 100 records, and must be formatted like this:

```
<Name Mask>,<Start1>,<Len1>,<Start2>,<Len2>,<Start3>,<Len3>
```

For example, this is a valid entry in x_palias.dat: ????PROC??,4,4,1,4,9,2

The "?" marks are wild card characters. In this case any process named like "????PROC??" will have its name altered to:

"substring(name,4,4) :: substring(name,1,4) :: substring(name,9,2)"

where "::" denotes a concatenation operation.

So the previous examples of conflicting names:

CUSTPROC01 (CUST = Customer)

OUSTPROC01 (OUST = Overseas Customer)

DUSTPROC01 (DUST = Domestic Customer)

would be altered to their alias values before they are subjected to the left truncation and invalid character removal algorithm:

Actual Alias Truncated

CUSTPROC01 PROCCUST01 OCCUST01

OUSTPROC01 PROCOUST01 OCOUST01

DUSTPROC01 PROCDUST01 OCDUST01

You could also control the truncation phase by using an x_palias.dat entry like this:

????PROC??,1,4,5,2,9,2

to produce non-conflicting names like this:

ActualAliasTruncatedCUSTPROC01CUSTPR01CUSTPR01OUSTPROC01OUSTPR01OUSTPR01DUSTPROC01DUSTPR01DUSTPR01

Before creating or editing x_palias.dat (in x_lansa) please read and observe the following rules and guidelines:

- This facility is provided to allow existing applications to run unchanged. Where duplicate names are encountered, please take immediate steps to ensure that no further (i.e.: new objects) are created that cause naming conflicts to occur. In the case of RUOM objects, a system flag setting may be used to ensure that shorter 8 character process names are generated. However, you should use extreme caution in deciding to turn this option on as it may cause all existing processes to be (re)generated by RUOM with the new (and different) 8 character names.
- When specifying X_PALIAS data check, and then double check, each entry very carefully. For performance reasons the routines that read x_palias.dat do almost no error checking. Invalid or badly formatted data may cause application failure and/or unpredictable results.

The format is:

<Name Mask>,<Start1>,<Len1>,<Start2>,<Len2>,<Start3>,<Len3>

where:

- <Name Mask> is from 1 to 10 characters. Imbedded blanks are not allowed. "?" wildcard characters are supported. The mask must be delimited from the <Start1> value by a single comma. Leading blanks are not supported.
- <Startn> is a substring start position in the range 1 to 10. Special value 0 may be used to indicate that a substring operation is not required.
- <Lenn> is a substring length in the range 1 to 10. l Special value 0 may be used to indicate that a substring operation is not required. The combination of a start position and length that is outside the length imposed by the length of the mask will cause unpredictable results and/or application failure.

There must be exactly 6 commas on each line delimiting the mask and 3 pairs of substring values exactly as shown.

There must be exactly 3 pairs of substring values separated by commas. Where a <Startn>,<Lenn> pair is not required use the special value 0. For example:

????PROC??,1,10,0,0,0,0

is valid and produces an alias identical to the starting name.

- There must be no more than 100 lines/entries in this file.
- The recommended standard entry for resolving RUOM generated process name conflicts is:

????PROC??,1,4,5,2,9,2

- x_palias.dat entries are processed from first to last until a match is found. When a match is found, processing terminates and no further comparisons are performed. Thus mask precedence is determined by the closeness of the mask to the start of the x_palias.dat file.
- When designing a sub-stringing algorithm you must avoid an approach that will result in conflicts with other objects such as 7 character RDML function names and 8 character OAM names. The substring approach you adopt and any results or problems that it causes, in any form whatsoever, are entirely your own responsibility.
- When you create or change the x_palias.dat file please note:
- You should manually delete any source, object or executable files that were created as a result of the previous (i.e. prior to your change) naming truncation/algorithm being used.
- You should force a regenerate/recompile of all the processes on your system

that will now have a new alias name produced as a result of the x_palias.dat changes you have made. Note that this point refers to processes - it does not refer to RDML functions.

- Remember to export x_palias.dat when moving/migrating your application to another system. The content of x_palias.dat is loaded into memory when you execute a Visual LANSA generated application via the x_run command.
- No matter what process naming/truncation method you use, the effect is only at the lowest level. There are no changes required in Visual LANSA or in RDML coding. At these levels the process is known by, and always referenced by, its full 10 character name.

Likewise, the name you always specify on the x_run command (in the PROC= parameter) is the full 10 character LANSA process name.

13.4 Automatic Renaming for SQL/ODBC

When working with SQL/ODBC tables, you should be aware of the following:

- 10 character names for libraries will be truncated on the left to 8 characters
- Special characters in table, index and view names (not 'A'->'Z', '0'->'9' or '_') are replaced according to position. Refer to Portability Considerations in *The Database*.
- Field names may be changed when the table is created. Refer to Portability Considerations in *The Data Dictionary*.

For example:

- A file with a library name of DC@DEMOLIB may be created in a collection named XDEMOLIB
- A file with a library name of FR@DTALIB may be created in a collection named R_DTALIB
- A field in a file called COMMENT may be added to the SQL/ODBC table with the name C_MMENT

Note: All dictionary and RDML references remain unchanged, i.e. you still refer to DC@DEMOLIB or FR@DTALIB or COMMENT. Where appropriate, Visual LANSA will automatically change to conform to the rules. Take note of the restrictions and avoid using these names for any new systems.
13.5 Accessing the Generated C Code

When a process or function is submitted for compilation, the developer has the option of saving the generated C code. If you are planning to port your application to another platform, you would want to save the C code generated; otherwise, you will generally not save the C code.

The C code will be saved in the source sub-directory of your x_ppp partition directory. The C code can be edited using any of the standard source line editors.

The C code generated is well structured and well documented so that it may be easily maintained.

Saving the C code can use a considerable amount of extra disk space. It is recommended that you do not keep the C code unless you are planning to port to another platform. The C code can be quickly regenerated if it is needed.

13.6 Handling Relative Record Numbers

Under the IBM i, when a record is added to a database file, it is automatically allocated a unique relative record number.

This number can be used for the fast access of records when the relative record number is known.

A record's relative record number may also change when a database file is reorganized.

No user intervention is required for any of this to happen.

Visual LANSA does not have the same concept of relative record numbers as the IBM i database.

Under Visual LANSA, this automatic allocation of a relative record number must be "emulated". The emulation of relative record numbers will allow your existing applications that use relative record numbers to execute without being changed.

You should be aware of the following:

- To achieve this, an extra column @@RRNO (defined in the LANSA repository) is included with every table definition.
- An index is created for the @@RRNO column in the SQL/ODBC table and this is used for relative record access to the table by RDML functions.
- Because of this required "emulation" of relative record numbers, data access by relative record number will not have the same performance improvements under Visual LANSA as it does under the IBM i, and there is a small performance impact on INSERT operations to assign a new relative record number.
- Relative record numbers uniquely identify a row within a table on the current platform. There is no association at all between relative record numbers on differing platforms/servers. The only way to uniquely identify a row across multiple platforms is by its "primary key".
- The use of the relative record number column as a Foreign key in another File is not recommended. The primary objection is that your application will not be portable to the System i. On the System i, the relative record number is not a data element in the file; its a physical offset on the disk. This offset must be re-arranged periodically as part of a well maintained system. This changes the relative record number of a data row. There is also the less important issue that even on non System i platforms,

the relative record number may need to be renumbered if all the numbers become used. Given that it provides for up to 15 digits worth of rows to be created, it is unlikely to be reached in any practical application for many years to come.

There are two methods of assigning the relative record number.

• **Method 1 - Set auto generation flag to Yes** This is the preferred method of assigning the relative record number. It is faster than Method 2 and is easier to administer as it is contained entirely in the database. Set the *Auto RRN Generation* flag set to Yes, to automatically assign the number.

• Method 2 - Use an external file

Note: This method of assigning the relative record number is for backward compatibility and may become obsolete in the future.

The number assigned to this column when a row is inserted into a table is allocated from a file created in the path specified by the RPTH= parameter of the x_run function.

Where the SQL/ODBC table resides on a server and is shared by multiple clients, so should the file nominated by the RPTH= parameter. This means that all users sharing the SQL/ODBC table(s) are also sharing the relative record number assignment files and thus allocating numbers from the same "bucket".

There is one file created for each SQL/ODBC table in this format: <Table Name>.RRN.

13.7 Renumbering Relative Record Numbers

You need to renumber the LANSA relative record number column when it has been created using the Auto RRNO attribute. That is, when the RRNO column is an Identity column.

The following SQL scripts are provided as an example of how to perform this renumbering task using SQL Scripts. It presumes that you are an experienced database administrator. These scripts are provided on an as is basis. No warranty is implied or given. Please test them thoroughly before using. It is strongly recommended that you backup your database before executing the script.

In these scripts **OwnerName** should be set to the LANSA library and **TableName** to the LANSA file name.

SQL Server

Use SQL Server Management Studio to connect to the database and enter this script modifying OwnerName and TableName appropriately. Also change the **USE** statement to the name of your database.

The statements are displayed as they are executed.

The script drops the RRNO index, drops the RRNO column from the table, adds the column back into the table (resetting the initial number to 1) and then recreates the index.

```
USE [MYDATABASE]
GO
BEGIN
SET NOCOUNT ON -- Allows concatenation to work
```

DECLARE @OwnerName varchar(max); DECLARE @TableName varchar(max); DECLARE @FullName varchar(max); DECLARE @IndexName varchar(max); DECLARE @SQL nvarchar(max);

```
SET @OwnerName = 'MYLIBRARY';
SET @TableName = 'MYFILE';
SET @FullName = (@OwnerName + '.' + @TableName)
-- PRINT @FullName
```

SET @IndexName = @TableName + '_R'; -- PRINT @IndexName

```
SET @SQL = 'DROP INDEX ' + @IndexName + ' ON ' + @FullName;
PRINT @SQL
EXECUTE sp_executesql @SQL;
```

SET @SQL = 'ALTER TABLE ' + @FullName + ' DROP COLUMN X_RR PRINT @SQL EXECUTE sp_executesql @SQL;

```
SET @SQL = 'ALTER TABLE ' + @FullName + ' ADD "X_RRNO" decim.
PRINT @SQL
EXECUTE sp_executesql @SQL;
```

```
SET @SQL = 'CREATE UNIQUE NONCLUSTERED INDEX ' + @Index!'
PRINT @SQL
EXECUTE sp_executesql @SQL;
```

END

SQL Anywhere

Connect to the database and enter this script modifying OwnerName and TableName appropriately.

The script drops the RRNO index, drops the RRNO column from the table, adds the column back into the table (resetting the initial number to 1) and then recreates the index.

BEGIN

DECLARE @@OwnerName char(128); DECLARE @@TableName char(128); DECLARE @@IndexName char(128);

SET @@OwnerName = 'MYLIBRARY'; SET @@TableName = 'MYFILE';

SET @@IndexName = STRING(@@TableName, '_R');

```
EXECUTE IMMEDIATE 'DROP INDEX ' || @@IndexName;
EXECUTE IMMEDIATE 'ALTER TABLE ' || @@OwnerName || '.' || @@Ta
EXECUTE IMMEDIATE 'ALTER TABLE ' || @@OwnerName || '.' || @@Ta
EXECUTE IMMEDIATE 'CREATE UNIQUE INDEX ' || @@IndexName ||
END
```

Oracle

Connect to the database and enter this script modifying OwnerName and TableName appropriately.

This script is different to the other two as we need to manually re-sequence; its not a feature of dropping a column and re-creating it.

DECLARE OwnerName varchar2(128) := 'SCOTT'; TableName varchar2(128) := 'ROB7_ORA'; FullName varchar2(128) := OwnerName || '.' || TableName; IndexName varchar2(128) := TableName || '_R'; FullIndexName varchar2(128) := OwnerName || '.' || IndexName; MaxRRNO NUMERIC := 0; NextSeq NUMERIC := 0; plsql VARCHAR2(500); BEGIN -- Remove Index so we can have duplicates whilst we re-number EXECUTE IMMEDIATE 'DROP INDEX ' || IndexName;

-- Re-sequence X_RRNO EXECUTE IMMEDIATE 'UPDATE ' || FullName || ' SET X_RRNO=ROWNU

-- Need to get the max RRNO back, so must use a PL/SQL Block plsql := 'BEGIN SELECT MAX(X_RRNO) INTO :a FROM ' || FullName || '; EXECUTE IMMEDIATE plsql USING IN OUT MaxRRNO;

```
DBMS_OUTPUT.PUT_LINE('Maximum RRNO = ' || MaxRRNO);
```

IF MaxRRNO IS NULL THEN NextSeq := 1; ELSE NextSeq := MaxRRNO + 1; END IF;

--- Reset sequence to Max EXECUTE IMMEDIATE 'DROP SEQUENCE ' || FullName || 'X_RRNO_SEC EXECUTE IMMEDIATE 'CREATE SEQUENCE ' || FullName || 'X_RRNO_S START WITH ' || NextSeq || 'INCREMENT BY 1';

-- Create Index now that we have unique numbers EXECUTE IMMEDIATE 'CREATE UNIQUE INDEX ' || FullIndexName || ' C

END;

13.8 U_TEXT Strings and .UTX Files

Whenever C code is generated for a Process, a Function or an OAM, referenced text strings are defined in a file with the same name as the object, followed by a txt suffix.

Within these .txt files is defined every text string as a Unicode value.

The Unicode form is used because it cannot be corrupted by page code variations when you transfer source code from platform to platform.

The actual conversion of the strings to hexadecimal form involves the use of automatic conversion tables provided by the operating system. This is performed as part of compiling on the target platform and produces a .UTX file which is compiled into the executable object.

13.9 The SLLLLOOO & DLLLLOOO.DAT Tables

Support for "sllllooo.dat" or "dllllooo.dat" has been removed.

13.10 Things That May Be Different

SQL and ODBC do not handle all of their table operations in the same manner as file operations on the IBM i. Here are some important points which you should be aware of:

- An OPEN RDML command under the IBM i physically opens the database file. Under Visual LANSA using SQL/ODBC however, an OPEN RDML command is a logical open only, as a physical database file open only occurs when actually performing an operation such as FETCH or UPDATE on the file. Likewise a CLOSE RDML command only logically closes the database file when using SQL/ODBC under Visual LANSA. Any applications that may rely on an error being returned from an OPEN or CLOSE RDML command should be reviewed (e.g.: to check if a file/table exists).
- On this SELECT loop point 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. They do not have predictable and consistent values outside the
loop. So:
```

```
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. Again, this type of logic would not be expected in sensibly constructed applications, so do not worry too much about this point. The value of A (and 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.

- SQL/ODBC indexes will only be created for logical view definitions that have access path "IMMED".
- Record locking is not supported. You should use the LOCK_OBJECT built in function instead.
- Alternate collating sequences for logical files are not supported.
- IBM i database dependencies e.g. multi-member files are not supported. Files will be implemented as SQL/ODBC tables and as such are made up of simple columns (fields) and rows (records). Additional indexes and views of the table may be created. It is suggested that your applications use files in this simple manner.
- SELECT/OMIT criteria for logical view definitions with all SELECT or all OMIT operations are fully supported **for row reading purposes only**. Mixing SELECT and OMIT operations may not produce the same results when using SQL/ODBC views as using the IBM i logical views and will require testing.
- The emulation facilities for the IBM i select/omit criteria are only provided to support the **reading** of selected information from database tables. They are not provided to (or designed to) allow you to control what information is inserted or updated into database table rows. If you are using an application design that relies on the use of uniquely keyed logical files (with associated select/omit criteria) to control the **content** of the your database then you may have to revise this component of your design.

13.11 ASCII Versus EBCDIC Data Collating Sequences

ASCII and EBCDIC have fundamentally different collating sequences. ASCII format is used on the Visual LANSA while EBCDIC is used by the IBM i.

For example, in EBCDIC the numbers '0' -> '9' are collated after the letters of the English alphabet. In ASCII, they are collated before the alphabet. Your data may now appear in a different sequence.

In sensibly designed modern applications, this should not affect the overall processing of an application, although it may affect the sequence that some individual applications present information.

13.12 Commitment Control

When working with commitment control, you should be aware of the following:

- All files are automatically under commitment control all of the time under Visual LANSA.
- Unless you indicate that a file is under commitment control (and therefore that COMMIT commands are issued by the application), an SQL/ODBC COMMIT will be issued after each I/O operation that updates the database.

This approach closely emulates the IBM i running a job without commitment control active, and it is very "clean and simple" to use. There are no known complications.

- As a result of duplicate key errors, some database management systems lock rows as though database changes have occurred. This type of error is returned as a validation error to an application. To release any such locks for files that are not under commitment control, LANSA automatically issues a COMMIT after a duplicate key error. For files that are under commitment control, it is the application developer's responsibility to determine whether the DBMS being used locks rows after a failed insert or update. If the DBMS does lock rows, you must issue a COMMIT or ROLLBACK so that these locked rows do not cause blocking for other users that may be attempting to access the same rows.
- Locks only timeout by default on IBM i. Refer to Lock Timeout in the *Technical Reference* for information on how you can emulate this feature for other platforms.
- In SQL, each transaction has an **Isolation Level**, which is similar to IBM's Lock Level for commitment control. The isolation level determines the degree to which the transaction is isolated from changes made by other connections to the database. Refer to ODBI Parameter <in the *Technical Reference* and your database's manuals for further information.
- The function options *PGMCOMMIT and *NOPGMCOMMIT are ignored for Visual LANSA and RDMLX code.
- You should revise your commitment control strategy with all this information in mind. Some things to think about are:
 - If you are going to design and build applications using Visual LANSA commitment control then you should acquire some knowledge of how it

works and differs from the IBM i implementation of commitment control.

- Despite the hype, database management systems exhibit inter and intra commitment control differences. The same may be said for row locking methods in general, and for how row locking and timeouts are affected by commitment control in particular.
- Avoid actually using "programmer controlled" COMMIT and ROLLBACK techniques whenever possible. This simplifies your applications and makes you less likely to be affected by different DBMS commit/rollback methods in the future.
- If your application uses mixed mode commitment control (ie. some tables under RDML level commitment control, some not) please refer to 13.12.1 Using Mixed Mode Commitment Control for instructions on how to make this work on platforms other than the IBM i.
- If you must use program level COMMIT and ROLLBACK then keep your use clean and simple. Some very important considerations, that apply to all platforms, and to the general use of commitment control in any situation are:
 - Keep it simple. Complex and elaborate commitment control strategies may look good on paper, but they often do not work in practice, and will become a maintenance nightmare in the future. Judge the cost of implementing, maintaining and fully testing the strategy versus the actual benefits it provides at the business level. Often the cost is very, very high and the benefit (as opposed to some simpler restore strategy) is very, very small.
 - Use simple and well defined transaction boundaries.
 - Avoid transaction boundaries that span interactions with the user via the user interface. You may be holding/locking all sorts of system resources while your end user is off having a cup of coffee.
 - Avoid complicated strategies that involve subroutines that update data and rely on their callers to do the commits. This approach will quickly turn into a maintenance nightmare. What happens when the subroutine is modified to call another subroutine that actually issues a commit?
 - If an UPDATE command includes BLOB or CLOB fields, the UPDATE is not a single operation at the database layer. Therefore, when the I/O command returns ER, it is possible that the non-LOB fields have been updated, but one or more LOB fields have not. Therefore, LANSA recommends executing a ROLLBACK (or ABORT) to return your

database to its previous state. However, please note that some database management systems do not rollback BLOB and CLOB data. Please refer to your database manuals for further details.

13.12.1 Using Mixed Mode Commitment Control

Enabling Mixed Mode Commitment Control allows Visual LANSA applications to emulate IBM i commitment control, so that if some files are under commitment control and other files are not under commitment control, the application behavior will closely emulate the behavior on an IBM i.

By default, the SQL/ODBC COMMIT issued for files not under commitment control after each I/O command also commits files under 13.12 Commitment Control, as a single database connection is used for updates to both kinds of files. When Mixed Mode Commitment Control is enabled, separate database connections are used to manage the two sets of files, so files under commitment control are only committed by an explicit COMMIT.

To enable Mixed Mode Commitment control, add the line "MIXED_MODE_COMMITMENT=YES" to the X_DBMENV.DAT file on the system where the OAMs are executed, under the appropriate database type. Refer to X_DBMENV.DAT in the *Technical Reference Guide* for further information. For example :

DATABASE_TYPE=MSSQLS DATABASE_DESCRIPTION=Microsoft SQL Server ...

MIXED_MODE_COMMITMENT=YES

If this line is not present exactly as shown then the new feature will not be enabled and all I/O commands will continue to function as they did before.

Changes to the MIXED_MODE_COMMITMENT setting are immediate. No regeneration or recompilation of applications is required when the MIXED_MODE_COMMITMENT setting for an environment is changed.

This feature may not be useable with the database SYNC_POINT optimization facility. Refer to SET_SESSION_VALUE in the *Technical Reference Guide* for information. LANSA therefore recommends these features are not used together.

Enabling Mixed Mode Commitment for the main database type also enables the feature for I/O operations to PC Other Files, regardless of their actual database type. Thus, if DBUT=MSSQLS and MIXED_MODE_COMMITMENT=YES has been specified for MSSQLS, any PC Other Files will also have mixed mode commitment enabled.

Batch control operations will be committed (or rolled back) in the same way that operations to their associated "owner" file are. Thus if file ITEM (that is,

the "owner" file) has batch control logic associated with it that updates file ORDER, then the batch control updates/inserts to the ORDER file :

- Will use the same database connection as is currently being used to update the ITEM file.
- Will never be themselves committed. Commitments to the ORDER file are actually done (or not done) according to the rules effecting the "owner" ITEM file. If the ITEM file commits then the ORDER batch controls commit, if the ITEM file does not commit, then neither do the ORDER batch controls.

13.13 SQL/ODBC Grammar: Keyword Conflicts

One of the interesting things about SQL/ODBC grammar is the use of keywords. Keywords are words reserved by the DBMS to define, manipulate and access database objects.

Some examples are the words CREATE, GRANT, TABLE, SELECT, ORDER and BY.

Generally you should avoid using keywords as object names.

For example, naming a table TABLE or a column in a table GRANT may cause you needless problems and complexities in your applications, essentially because the SQL/ODBC command:

SELECT GRANT FROM TABLE

may upset many SQL/ODBC grammar parsers. Strictly speaking you can use a SQL/ODBC grammar format known as double quote delimited strings and do this:

SELECT "GRANT" FROM "TABLE"

However, the whole concept of using DBMS object names that conflict with DBMS keywords should best be avoided because the degree and consistency of support for the double quoted strings facility varies from DBMS to DBMS.

In fact a *Microsoft SQL Server* 6.0 *Guide* makes this recommendation:

"Even though it is syntactically possible, using keywords as objects names (with quoted identifiers), it is not recommended."

Visual LANSA has adopted the approach of not using conflicting object name/keyword identifiers by:

- quoting identifiers that may be SQL keywords when generating SQL statements, and where necessary,
- automatically (and transparently) renaming DBMS objects that conflict with DBMS reserved keywords.

Refer to Portability Considerations in *The Data Dictionary* and Portability Considerations in *The Database* for details on when DBMS objects names are converted. Obviously this facility was primarily provided for the upward compatibility of existing applications. Whenever possible you should avoid using field/column, file/table or library/collection names that conflict with DBMS reserved keywords in any **new applications** that you create. This means that if you define a physical file (that is, a table) named TABLE, then the following will happen:

- When Visual LANSA creates the table it will automatically rename it to T_BLE so that there is no object name/keyword conflict at the DBMS level.
- The name TABLE can be used throughout LANSA and it will be automatically and transparently converted to T_BLE as required.

13.13.1 Reserved Keywords – SQL/ODBC

The following list shows the reserved keywords for field and file identifiers that may be renamed by LANSA. These keywords are from the reserved words lists for Microsoft's SQL Server, Sybase's SQL Anywhere and Oracle's Database Server.

@@DBTS	DATATYPE	INCLUDE	OTHERS	SIZE
@@ERROR	DATE	INCREMENT	OUT	SKIP
@@LANGID	DATEADD	INDDN	OUTER	SMALLI
@@SPID	DATEDIFF	INDEX	OUTPUT	SOME
ABSOLUTE	DATENAME	INDEXES	OVER	SORTDE
ACCESS	DATEPART	INDICATOR	OVERLAPS	SORTED
ACQUIRE	DATETIME	INFILE	PACKAGE	SORTNL
ACTION	DAY	INITIAL	PAD	SOUNDI
ADA	DAYS	INITIALLY	PAGE	SPACE
ADD	DBA	INNER	PAGES	SQL
ADDRESS	DBADM	INOUT	PARALLEL	SQLBUF
AFTER	DBCC	INPUT	PART	SQLCA
AGGREGATE	DBCS	INPUT	PARTIAL	SQLCOI
AHEAD	DBSPACE	INSERT	PASCAL	SQLERR
ALIAS	DEALLOCATE	E INSTALL	PATINDEX	SQLSTA
ALL	DEC	INSTEAD	PCTFREE	SQLWAI
ALLOCATE	DECIMAL	INT	PCTINDEX	SQRT
ALTER	DECLARE	INTEGER	PENDANT	SQUARI
ANALYZE	DEFAULT	INTERSECT	PERCENT	START
AND	DEFAULTIF	INTERVAL	PERM	STATEM
ANY	DEFERRABLE	E INTO	PERMANENT	STATIST
APPEND	DEFERRED	IS	PIECED	STDEV
ARE	DELETE	ISDATE	PIPE	STDEVP
ARRAY	DELIMITED	ISNULL	PIVOT	STOGRC
ARRAYLEN	DENY	ISNUMERIC	PLAN	STOP
AS	DEPTH	ISOLATION	PLI	STORPC

ASC	DEREF	JOIN	POSITION	STR
ASCII	DESC	KEY	PRECISION	STREAN
ASENSITIVE	DESCRIBE	KEY_GUID	PREPARE	STRIP
ASSERTION	DESCRIPTOR	KEY_ID	PRESERVE	STUFF
ASYMKEY_ID	DICTIONARY	KEY_NAME	PRIMARY	SUBSCR
ASYMMETRIC	DIFFERENCE	KILL	PRINT	SUBSTR
AT	DISABLE	LABEL	PRIOR	SUBSTR
ATOMIC	DISCARDDN	LABELS	PRIQTY	SUBTRA
AUDIT	DISCARDS	LANGUAGE	PRIVATE	SUCCES
AVERAGE	DISCONNECT	LAST	PRIVILEGES	SUM
AVG	DISK	LEADING	PROC	SYNON
BACKUP	DISTINCT	LEAVE	PROCEDURE	SYSDAT
BADFILE	DO	LEFT	PROGRAM	SYSTEM
BDDN	DOMAIN	LENGTH	PUBLIC	TABLE
BEFORE	DOUBLE	LEVEL	PUBLISH	TABLEC
BEGIN	DROP	LIKE	PUBLISHER	TABLES
BEGINDATA	DUAL	LINENO	PURGE	TAPE
BETWEEN	DUMMY	LIST	PUT	TEMP
BINARY	DUMP	LOAD	QUOTENAME	TEMPOI
BIND	DURATION	LOCAL	QUOTES	TERMIN
BINDADD	DYNAMIC	LOCK	RAISERROR	TEST
BIT	EACH	LOCKSIZE	RAND	TEXT
BIT_LENGTH	EDITPROC	LOG	RAW	TEXTPT
BLANKS	ELSE	LONG	READ	TEXTSIZ
BLOB	ELSEIF	LOOP	READTEXT	TEXTVA
BLOCK	ENABLE	LOWER	REAL	THEN
BLOCKSIZE	ENCLOSED	LTRIM	RECEIVED	THIS
BOOLEAN	ENCRYPTED	MATCH	RECLEN	TIME
BOTH	END	MAX	RECNUM	TIMEST
BREADTH	END_EXEC	MAXEXTENTS	RECOMPILE	TINYIN
BREAK	END-EXEC	MEMBERSHIP	RECORD	ТО

BROWSE	ENDIF	MERGE	REFERENCE	ТОР
BUFFERPOOL	ERASE	MESSAGE	REFERENCES	TRAILIN
BULK	ERRLVL	MIN	RELATIVE	TRAN
BY	ERROR_LINE	MINUS	RELEASE	TRANSL
С	ERROREXIT	MINUTE	REMOTE	TRIGGE
CACHE	ESCAPE	MINUTES	REMOVE	TRIM
CALL	ESCAPES	MIRROREXIT	RENAME	TRUE
CALLED	EVERY	MIXED	REOPEN	TRUNC
CASCADE	EXCEPT	MODE	REPEATABLE	TSEQUA
CASE	EXCEPTION	MODIFY	REPLACE	TYPE
CAST	EXCLUSIVE	MODULE	REPLICATE	UID
CATALOG	EXEC	MONTH	RERECEIVE	UNICOE
CERT_ID	EXECUTE	MONTHS	RESEND	UNION
CHAR	EXEMPT	Ν	RESET	UNIQUE
CHARACTER	EXISTS	NAME	RESIGNAL	UNITS
CHARINDEX	EXIT	NAMED	RESOLVE	UNKNO
CHECK	EXPLAIN	NAMES	RESOURCE	UNLOAI
CHECKPOINT	EXTEND	NATIONAL	RESTORE	UNLOCI
CHECKSUM	EXTENDED	NATURAL	RESTRICT	UNPIVO
CLASS	EXTERNAL	NCHAR	RESULT	UNTIL
CLOB	EXTRACT	NETWORK	RESUME	UPDATE
CLOSE	FALSE	NEW	RETURN	UPDATE
CLUSTER	FETCH	NEXT	RETURNS	UPPER
CLUSTERED	FIELDPROC	NHEADER	REVERSE	USAGE
COALESCE	FIELDS	NO	REVERT	USE
COBOL	FILE	NOAUDIT	REVOKE	USER
COLLATE	FILLFACTOR	NOCHECK	RIGHT	USING
COLLATION	FIRST	NOCOMPRESS	ROLLBACK	VALIDA
COLLECT	FIXED	NOHOLDLOCK	ROUND	VALIDP]
COLUMN	FLOAT	NONE	ROW	VALUE
COMMENT	FLOPPY	NOT	ROWCOUNT	VALUES

COMMIT	FOR	NOTFOUND	ROWGUIDCOL	VAR
COMMITTED	FOREIGN	NOTIFY	ROWID	VARBIN
COMPLETION	FORMAT	NOWAIT	ROWLABEL	VARCH
COMPRESS	FORTRAN	NULL	ROWNUM	VARCHA
COMPUTE	FOUND	NULLCOLS	ROWS	VARGRA
CONDITION	FRACTION	NULLIF	RTRIM	VARIAB
CONFIRM	FREETEXT	NUMBER	RULE	VARIAB
CONNECT	FROM	NUMERIC	RUN	VARP
CONNECTION	FULL	NUMPARTS	SAVE	VARYIN
CONSTANT	FUNCTION	NVARCHAR	SAVEPOINT	VCAT
CONSTRAINT	GENERATED	OF	SBCS	VERIFY
CONTAINS	GET	OFF	SCHEDULE	VIEW
CONTINUE	GETDATE	OFFLINE	SCHEMA	VOLUM
CONTINUEIF	GETUTCDATE	OFFSET	SCROLL	WAIT
CONTROL	GLOBAL	OFFSETS	SECOND	WAITFO
CONTROLROW	GO	OLD	SECONDS	WHEN
CONVERT	GOTO	ON	SECQTY	WHENE
CORR	GRANT	ONCE	SECTION	WHERE
COUNT	GRAPHIC	ONLINE	SELECT	WHILE
COUNT_BIG	GROUP	ONLY	SEND	WHITES
COVAR_POP	GROUPING	OPEN	SENT	WITH
COVAR_SAMP	HAVING	OPENQUERY	SEQUENCE	WORK
CREATE	HOLD	OPENROWSET	SESSION	WORKD
CREATETAB	HOLDLOCK	OPENXML	SET	WRITE
CROSS	HOUR	OPTIMIZER	SETTING	WRITET
CUBE	HOURS	OPTION	SETUSER	YEAR
CURRENT	IDENTIFIED	OPTIONALLY	SHARE	YEARS
CURSOR	IDENTITY	OPTIONS	SHUTDOWN	YES
CYCLE	IF	OR	SIGNAL	ZONE
DATA	IMMEDIATE	ORDER	SIGNBYCERT	ZONED
DATABASE	IN			

13.14 General ODBC Driver Limitations

ODBC drivers may impose limitations on the SQL grammar if single-tier configuration is used, e.g. a driver for ISAM file. Otherwise in the multiple-tier configuration the database server will process the SQL.

Some examples of the limitations are:

- The NOT NULL constraint in the CREATE TABLE statement may not be not supported.
- A maximum of 40 AND predicates is supported.
- The CREATE INDEX statement is not supported for Microsoft Excel or Text drivers.
- Views are not supported by the Btrieve, dBASE, Microsoft Excel, Microsoft FoxPro, Paradox or Text driver.

14. Portability Specifics

14.1 Use of Hex Values, Attributes, Hidden/Embedded Decimal Data, *LOVAL and *HIVAL

- 14.2 IBM i Emulated Features
- 14.3 Calling 3GL Programs / DLLs / Shared Objects
- 14.4 3GL Virtual Code
- 14.5 Additional Notes
- 14.6 User Defined Messages

14.1 Use of Hex Values, Attributes, Hidden/Embedded Decimal Data, *LOVAL and *HIVAL

When working with hexadecimal values, attribute bytes, "hidden" or "imbedded" decimal data, *LOVAL or *HIVAL you should be aware of the following:

- Literal hexadecimal values (i.e. X'..') are not supported anywhere in Visual LANSA, in any form.
- Applications that imbed IBM i attribute bytes in data will produce unpredictable results under Visual LANSA.
- The use of special values *LOVAL (i.e. hexadecimal 00 (under IBM i) and hexadecimal 01 (other platforms)) or *HIVAL (i.e. hexadecimal FF on all platforms) in any data (in any form) is not recommended. Note that under IBM i *LOVAL is interpreted as X'00', whereas on all other platforms it is interpreted as X'01'. In both cases the hexadecimal value represents the lowest possible value that can be sensibly placed into an alphanumeric field.
 Generally *LOVAL and *HIVAL will function normally. However, failing to

Generally *LOVAL and *HIVAL will function normally. However, failing to observe these guidelines may produce unpredictable results:

- Avoid using them in new applications.
- Do not place fields containing them into database tables.
- Do not place fields containing them into reports.
- Do not place fields containing them onto screen panels.
- Do not use them in fields exchanged or passed between differing platforms (e.g. CALL_SERVER_FUNCTION).
- Do not use them as values used to access data in a DBMS
- The "hiding" of decimal data (packed or signed) inside alphanumeric fields is not a recommended technique. The Field Attribute *SBIN removes some restrictions and is described in the next point.
 Where this technique is used with message substitution variables unpredictable results will occur.
 Where this technique is used with database fields (e.g. an alphanumeric field of length 48 is actually a storage area for 12 packed decimal (7,0) numbers)
 - predictable results will always occur:
- If the data is created on the current platform

- If the data always stays on the current platform.
- If the DBMS system you are using supports the storage of imbedded X'00' values in character or variable character fields (many do not support this). In other words, hiding decimal data inside alphanumeric fields may work as expected until some sort of data transfer, data storage, client/server, query or reporting application is used.

At this stage, the whole "hiding" approach will totally collapse and the data will effectively become rubbish.

If you have an application that makes heavy use of numeric data hidden inside alphanumeric fields then use the attribute *SBIN as described in the next point.

• The input and output Field attribute *SBIN can be used on an alphanumeric field to indicate that the database must not perform any character conversions on it. It only makes a difference on non-IBM i platforms as the IBM i database does not perform any character conversion on alphanumeric fields. On non-IBM i databases, when the client's code page is different to the server's code page, the database will need to convert between the two code pages. This conversion is usually not bi-directional for all characters in the character set. That is, a character entered as ¥ may be returned as Ñ.

This is most commonly encountered with non-English and non-Japanese SQL Anywhere databases when a Translator is used in the ODBC data source. Refer to the *Sybase SQL Anywhere Manual* for more information on the use of a Translator. Do not change this option to attempt to make your embedded packed fields work as other more significant problems will be introduced.

The issue can also occur with, say, an SQL Server database running on an English Windows server and clients running on Greek Windows.

Using the *SBIN attribute on either input, output or both, causes the field to be created as a column of type binary in the database or the equivalent, for example, RAW in Oracle. All database reads and writes will just copy the actual data and not presume anything about the content. For example, embedded X'00' characters would be faithfully written and read from the database without causing any following data to be truncated.

It is important to remember that this functionality is only true at the database

level. Manipulating an alphanumeric field with embedded packed fields as an alphanumeric field, rather than its packed constituent parts, will still yield unpredictable results as described in the previous point. In RDML you must refer to the embedded fields and not the alphanumeric field.

This is an example of the code: FUNCTION OPTIONS(*DIRECT); DEFINE FIELD(#A04) TYPE(*CHAR) LENGTH(4); DEFINE FIELD(#A04A) TYPE(*CHAR) LENGTH(4) DECIMALS(0) TO_O DEFINE FIELD(#P03R) TYPE(*DEC) LENGTH(003) DECIMALS(0) EDIT DEFINE FIELD(#C) TYPE(*DEC) LENGTH(007) DECIMALS(0); DEF LIST NAME(#L10) FIELDS(#P07 #P03R); ********* COMMENT(Insert 0 to 999 fields); BEGIN LOOP USING(#C) TO(1000); CHANGE FIELD(#P07) TO('(#C - 1)'); CHANGE FIELD(#P03R) TO(#P07); INSERT FIELDS(#P07 #A02) TO FILE(F10); END LOOP: ********* COMMENT(Retrieve and converts previously saved fields); SELECT FIELDS(#P07 #A02) FROM FILE(F10); ADD ENTRY TO LIST(#L10); **ENDSELECT:** ******

DISPLAY BROWSELIST(#L10);

- P07 is defined in the repository as packed, length 7 and decimals 0. A02 is defined as alphanumeric of length 2. These two fields are in file F10.
- Note that A02 is not manipulated directly in the RDML. Rather, P03R is manipulated in the RDML and A02 is only referred to in database commands like INSERT and SELECT. If, for example, A02 was used in a CHANGE command the behavior is unspecified.

CHANGE FIELD(#P03R) TO(#P07); CHANGE FIELD(#A04) TO(#A02)

It may or may not work as expected depending on whether the Packed Field representation #P03R (i.e., #A02) contains a byte of 0x00.

• *SBIN has particular restrictions that are not usually true for alphanumeric fields.

Uses that are not supported include:

- Displaying the Alphanumeric Field
- Using the Field in a File's Key
- Using the Field in a View's Key
- Using the Field in an Access Route's Key
- Using the Field in a Batch Control Key
- Using the Field in a Rule or Trigger
 - These uses are not enforced by the Repository. They may work on some databases and not on others. Because of this different database behavior it is advisable not to use binary fields in this way.

14.1.1 The Meaning of *NULL

The value *NULL is blanks for alphanumeric fields and zeroes for numeric fields. It is not NULL (i.e. X'00') in the "C" language context.

14.2 IBM i Emulated Features

IBM i data areas may be emulated in an SQL table called LX_DTA in the LX_DTA collection or in flat files. For details, refer to the DASO= Parameter in Standard X_RUN Parameters (*Using the X_RUN Command*), in the *Technical Reference Guide*.

14.2.1 IBM i Defined Data Areas

Before attempting to execute applications that manipulate data areas via the associated "get" and "put" Built-In Functions, please note:

- Data areas will be automatically enrolled in the LX_DTA table when the first attempt is made to access a non-enrolled data area.
 Note: The data area locking/unlocking logic actually uses the LOCK_OBJECT and UNLOCK_OBJECT functions. A "Data area not found" error may leave the data area locked in the locking table. To remove this lock use a small function that uses the UNLOCK_OBJECT, or use a DBMS direct access tool to remove the lock row from table LX_FOL.
- Alternatively, you may manually enroll the Data Area in the LX_DTA table by using a DBMS supplied direct access tool. Specify the library name as *LIBL unless you have an application that uses specific library names.

Specify the data area type as "A" (alphanumeric) or "N" or "D" (numeric) when enrolling it in table LX_DTA. Note that both *AUTONUMxxxxxxxx and *AUTOALPxxxxxxxx data areas should be enrolled as "N" or "D" (numeric) values. Even though *AUTOALPxxxxxxxx values are ultimately mapped into alpha fields they are actually a numeric value.

- The truncation of numeric data area values may differ from the IBM i implementation. In a poorly designed application setting a 4 digit numeric data area from a 5 digit numeric field that contains 54321 would set the data area value to 4321. Under Visual LANSA, the data area value would be set to 54321. This is unlikely to cause a problem because most applications should be implemented to handle the "rollover" point themselves.
- The "hiding" of packed decimal data in alphanumeric data areas is not supported.
- Numeric data areas can be up to 24,9 precision.

14.2.2 *AUTONUM and *AUTOALP System Variables (Data Areas)

IBM i data areas are emulated in an SQL table called LX_DTA in the LX_DTA collection.

Before attempting to execute applications that use *AUTONUM or *AUTOALP system variables and the data areas behind them, please note the following:

• *AUTONUM and *AUTOALP system variables will be automatically enrolled in the LX_DTA table when the first attempt is made to use one of them.

The data area locking/unlocking logic actually uses the LOCK_OBJECT and UNLOCK_OBJECT functions. A "Data area not found" error may leave the data area locked in the locking table. To remove this lock use a small function that uses the UNLOCK_OBJECT, or use a DBMS direct access tool to remove the lock row from table LX_FOL.

 Alternatively, *AUTONUM and *AUTOALP system variable data areas may be manually enrolled in the LX_DTA table. If you do this, always specify the library name as *LIBL. The data area name is derived from the *AUTOALP/*AUTONUM system variable name. For example, *AUTONUM07NXTPROD would be enrolled with the name NXTPROD.

Specify the data area type as "N" or "D" (numeric) for both *AUTONUM and *AUTOALP data areas. Even though the RDML level reference may be alpha (*AUTOALP), the actual data area behind the system variable is actually numeric.

• *AUTONUM/ALP values "roll over" when their maximum is reached (exactly as they do under IBM i). So *AUTONUM07NXTPROD rolls over after the value 99999999 is used.

Note: *DTAssslllxxxxxxxx, *AUTONUMnnxxxxxxx and *AUTOALPnnxxxxxxxx system variable evaluation requests can be automatically diverted into a server system when locks are diverted into a server system. Refer to the DEFINE_ANY_SERVER Built-In Function description in the *Technical Reference Guide* for more information.

14.2.3 *DTASSSLLLXXXXXXXXX System Variables (Data Areas)

*DTASSSLLLXXXXXXXXXX

where

Sss is the starting position in data area

lll is the length to retrieve

Xxxxxxxx is the data area name

IBM i data areas are emulated in an SQL table called LX_DTA in the LX_DTA collection.

Before attempting to execute applications that use *DTAssslllxxxxxxxx system variables and the data areas behind them, please note the following:

• *DTAssslllxxxxxxxx system variables will be automatically enrolled in the LX_DTA table when the first attempt is made to use one of them.

The data area locking/unlocking logic actually uses the LOCK_OBJECT and UNLOCK_OBJECT Built-In Functions. A "Data area not found" error may leave the data area locked in the locking table. To remove this lock use a small function that uses the UNLOCK_OBJECT Built-In Function, or use a DBMS direct access tool to remove the lock row from table LX_FOL.

- Alternatively, *DTAssslllxxxxxxxx system variable data areas may be manually enrolled in the LX_DTA table. If you do this, always specify the library name as *LIBL. The data area name should come from the name component of the *DTAssslllxxxxxxxx name. For example the system variable *DTA010005COMPINF would be enrolled as a data area called COMPINF in library *LIBL.
- Specify the data area type as "A" (alphanumeric) in all cases.
- The "hiding" of packed decimal data in alphanumeric data areas is not supported.

Note: *DTAssslllxxxxxxxx, *AUTONUMnnxxxxxxx and *AUTOALPnnxxxxxxxx system variable evaluation requests can be automatically diverted into a server system when locks are diverted into a server system. Refer to the DEFINE_ANY_SERVER Built-In Function description in the *Technical Reference Guide* for more information.

14.2.4 Emulation of the *LDA Data Area

The IBM i *LDA (Local Data Area) is supported.

The use of the IBM i *LDA type of data area in new applications is not recommended. This feature is provided for support of existing applications and is not considered to be a "strategic" facility.

By its nature, the LDA is unique to every job and thus does not have to be enrolled in the data area definition table LX_DTA.
14.2.5 Data Areas and Other LANSA Features

It is important to remember that other LANSA features use data areas. These features include the following:

- Some system variables (e.g. *COMPANY)
- Some Built-In Functions (e.g. SAVE_LIST).

If you are using other LANSA features which use data areas, you must remember to create the data area in the LX_DTA table.

14.2.6 The DC@A08 Data Area and the SAVE_LIST Built-In Function

The SAVE_LIST (Save Working List) Built-In Function may automatically allocate a name for the list to be saved.

Under IBM i the name is allocated from the content of a data area named DC@A08. Under Visual LANSA the name is also allocated from a data area named DC@A08.

Note: The data area locking/unlocking logic actually uses the LOCK_OBJECT and UNLOCK_OBJECT Built-In Functions. A "Data area not found" error may leave the data area locked in the locking table. To remove this lock use a small function that uses the UNLOCK_OBJECT Built-In Function, or use a DBMS direct access tool to remove the lock row from table LX_FOL.

Thus a data area named DC@A08 of type "N" (numeric) must be enrolled in the LX_DTA Data Area table before attempting to use the SAVE_LIST Built-In Function.

14.2.7 Guidelines, Rules and Limitations that Apply when Using the DASO=F Parameter

- Generally this option is faster and uses less system resources than the DASO=D option.
- This option causes the data area value to be stored in a file named <data area>.ARA in the same directory as the relative record number assignment files (.RRN) are stored. Refer to the RPTH= parameter for details of how this directory name is defaulted and specified. The file names used are reflective of the uppercase data area name, so data area names of more than 8 characters will cause file names of more than 8 characters to be created.
- You must ensure that all data area names that are used conform to IBM i object naming standards. This means that the first character must be one of A ->Z (uppercase), \$, @ or # followed by up to 9 more characters that must be one of A ->Z (uppercase), 0 -> 9, \$, @ or #. No other characters are permitted. Failure to observe this rule may lead to application failure and/or unexpected results.
- Locking is accomplished by using the appropriate operating system file access facilities. This means that data areas that are left locked (e.g. via the GET_CHAR_AREA Built-In Function) leave an operating system file handle open until such time as they are unlocked or the application terminates. Some operating systems impose a limit on the number of open file handles that you can have at any one time and that may limit how many locked data areas you can actually have concurrently locked. It is recommended that you do not try to have more than 4 concurrently locked data areas.
- The use of the option to divert data area accesses to a server system when running applications in SuperServer mode takes precedence over this option.

Normally only standalone systems and server systems use this option. Clients connected by LANSA SuperServer facilities do not normally use this option, relying instead on the SuperServer connection option to divert their data area access requests to the server system automatically. The server system itself then may choose to use DASO=F to further divert incoming client requests into flat files rather than the DBMS table LX_DTA.

• Use caution when switching between the DASO=D (the default) and DASO=F options. Those data area values stored in table LX_DTA and those

stored in <data area>.ARA files must have their current values aligned each time such a switch occurs. The value alignment process is your responsibility.

14.2.8 Array Index Values and Process/Overlay Handling

Array index values and field overlays are the possible source of uncontrolled program crashes (i.e. not handled by standard Visual LANSA error handling).

On the IBM i, LANSA generates RPG code that is translated into an intermediate machine language. The intermediate machine language actually implements high speed array index boundary checking (at run time).

Visual LANSA, however, generates C code and C compilers do not typically generate array index boundary checking.

If you get this type of error, run the function again using the Visual LANSA debugger in animate mode until the point of failure. At this point, use the Show Fields pull down option and check that all array indexes are valid.

14.3 Calling 3GL Programs / DLLs / Shared Objects

To call an existing 3GL developed DLL or Shared Object under Windows refer to Creating your own Built-In Functions with Visual LANSA.

User defined Built-In Functions are the recommended method of accessing existing 3GL DLL or Shared Objects because:

- The interface between the 3GL object and the RDML function is clearly defined into the LANSA repository.
- The well defined interface is much easier to understand and to use. This aids application developer productivity. For example, Visual LANSA can prompt the developer for the arguments required by the interface.
- The well defined interface is much easier to port to other platforms in the future.
- The well defined interface supports advanced features such as optional arguments, return values and defaults.
- The use of a single well defined interface encourages the reuse of existing facilities and the use of modular design techniques.

14.4 3GL Virtual Code

When working with virtual field definitions which use 3GL code, you should be aware of the following:

- 3GL virtual code must be re-specified in ANSI C code. This may coexist with the 3GL RPG code.
- "C" virtual code is identified by placing a character "C" in the first column of the virtual field source code to be included when generating with Visual LANSA.
- Virtual code specified as C should only ever manipulate fields in their "vector" format. A good understanding of the "vector" storage format for fields and the functions available to manipulate them is needed prior to attempting to use virtual code.

Any virtual fields defined using the standard LANSA extended field definitions are supported.

14.5 Additional Notes

When working with Visual LANSA applications, you should be aware of the following considerations and recommendations:

- DO NOT introduce any IBM i dependencies e.g. multi-member files, data queue access.
- Any sort of "trick" or IBM i dependency (e.g. EOFDLY) will not be supported.
- The library list concept will not be supported in any way.
- Any method of passing any form of information or control values (e.g. cursor location) between functions except by PARM, PASS_DS and PASS_LST, *LDA access and EXCHANGE option to exchange all fields in a function on an F4=Prompt is not recommended.
- All existing LANSA/GUI rules apply to all screen panels:

OPTIONS(*OVERLAY) and OPTIONS(*NOREAD) for the DISPLAY and REQUEST commands are not supported.

All fields in a DISPLAY/REQUEST command must fit on a single panel (i.e. multiple screen formats not supported).

Fields and text cannot span lines in a panel.

Help text held in Office documents are not supported.

14.6 User Defined Messages

- These should be placed into a user defined message file rather than DC@M01.
- If you add user defined messages to DC@M01, the messages may not be available at execution time. This will depend on how you deploy your application.
- A specific set of run time messages are copied from DC@M01 by *SYSEXPORT or when "Deploy to client without local database" is selected in the Deployment Tool. This is done in order to reduce the size of the extracted files.

15. LANSA Limits

- Maximum BLOB/CLOB file size:2³¹ 1 bytes (2 GB)
- Maximum number of rows per database table: 10¹⁶ 1 rows
- Maximum row size: 32,767 bytes. Does not include String and Binary columns. Limit effects the UNLOAD_FILE_DATA BIF
- Maximum static working list total size: IBM i 16 MB; all other platforms 2³¹
 1 bytes (2 GB)
- Maximum static working list entry width: IBM i 16 MB; all other platforms limited by available memory.
- Maximum dynamic working list total size: limited only by available memory
- Maximum dynamic and static working list entrys: 2³¹ 1
- Maximum dynamic working list entry width: IBM i 16 MB; all other platforms 2³¹ 1 bytes (2 GB), not including String and Binary field widths which are external to the entry width.
- Maximum external operating system file size: 2³¹ 1 bytes (2 GB). Affects these Built-In Functions:
 - UNLOAD_FILE_DATA
 - LOAD_FILE_DATA
 - Stream File BIFs
 - TRANSFORM_FILE
 - TRANSFORM_LIST.
- Maximum size of a Space entry: 2³¹ -1 bytes.
- Maximum entrys in a Space: limited by available memory.

Note that these limits are the absolute maximum that LANSA may address. Typically the achievable limits are less depending on the particular operating system or database used or the configuration of the computer on which LANSA is running. In other words, just because this is the theoretical maximum, it does not mean that it is practical for the purpose to which you put it.

16. Internationalization with Unicode

16.1 Unicode16.2 Developing New16.3 Modifying ExistingSupport in
LANSAApplications with UnicodeApplications to use Unicode

What is Unicode?

Unicode is a universal text encoding standard. Rather than being just another way to encode written language, the Unicode standard seeks to gradually replace all other character encoding standards as they will eventually no longer be needed.

Unicode assigns each written character, across all languages, a unique number. As a point of reference, the first 128 characters in Unicode are exactly the same as the ASCII code table characters with the same code point values. And the first 256 characters are exactly the same as the ISO 8859-1 (ISO Latin-1) standard.

A character's unique number (also known as its code point) is written long hand as U+nnn where nnnn is a hexadecimal number. The Unicode Standard encodes characters in the range U+0000...U+10FFFF. Because this amounts to a 21-bit code space, there are a variety of ways to encode the actual numbers and streams of numbers. These encoding schemes are known as Unicode Transformation Formats (UTFs).

Common UTFs include UTF-8 (CCSID 1208 on IBM i) and UTF-16 (CCSID 1200 on IBM i). UCS-2 (CCSID 13488 on IBM i) is an older UTF and is a subset of UTF-16.

LANSA uses whichever UTF is appropriate. For example, UTF-8 is generally used for flat files for two reasons. The first is that only one byte is required for each character that is in the ASCII code table, so space is saved. The second reason is that there are no issues with byte order. A UTF-8 file is binaryidentical whether on Windows, IBM i, or Linux, so it can be easily copied from one platform to another.

This table gives some examples of different characters and how they are encoded.

Name	Latin Capital A	Greek Capital Sigma	CJK Unified Ideograph 2020F
Glyph	Α	Σ	•
Code Point (decimal value)	65	931	131599
U+nnnn	41	U+03A3	U+2020F
UTF-8	41	CEA3	F0A0888F
UTF-16 Intel	4100	03A3	D840DE0F
UTF-16 IBM i	0041	A303	40D80FDE

Why is Unicode Necessary?

Prior to the development of Unicode as a standard, data conversion from language to language was onerous, with data loss whenever a character could not be represented in a particular code page. It was difficult to correctly handle data that was outside of the norm for a particular region. For example, Japanese names could not be correctly stored or displayed in a system designed for English names, as the character sets and code pages used varied so widely between the two languages.

The introduction of Unicode allows every written language's characters to be handled correctly with no concern of data loss when the data is moved between different environments.

With Unicode, LANSA can now allow display and editing of any number of languages at the same time, both in the Visual LANSA IDE, and also in developed applications.

New development can take advantage of 16.1.1 Unicode Field Types from the beginning, thus minimizing future changes to support new languages or non-standard descriptive text such as names and addresses.

Unicode can also be utilized when modifying existing applications. Fields used in RDMLX applications can be changed to use one of the Unicode Field Types.

16.1 Unicode Support in LANSA

The following LANSA features utilize Unicode:

- 16.1.1 Unicode Field Types
- 16.1.2 Other Files
- 16.1.3 Intrinsic Functions
- 16.1.4 BIFs and Unicode
- 16.1.5 Web Application Modules (WAMs)
- 16.1.6 Multilingual Descriptions
- 16. Internationalization with Unicode

16.1.1 Unicode Field Types

- The Unicode field types are Nchar and Nvarchar described in the *Technical Reference Guide*. These types are analogous to Char and String described in the *Technical Reference Guide*.
- The Unicode field types allow data to be handled as Unicode throughout an RDMLX application, in the UI and in the database. This means an application can allow data from multiple languages to be displayed and edited, saved in/read from the same database table, or sent to and received from external servers, all without data loss.
- Existing Alpha, Char, and String fields may be changed to Nchar or Nvarchar to enable your existing application to support world-wide multilingual customer data (such as names and addresses) as well as language-specific descriptions for code tables and many other situations. Refer to 16.3 Modifying Existing Applications to use Unicode for more information.
- Nchar and Nvarchar can be displayed and edited on Components and WAMs. Both Unicode field types are supported across LANSA products such as LANSA Integrator, LANSA Client and LANSA Open for .NET.
- Nchar and Nvarchar are handled differently from Char and String with the SUNI attribute. The SUNI attribute indicates that the field is Unicode in the database layer only. Elsewhere, it is handled in the current code page, which means that if you assign an Nchar to a Char with SUNI, data loss may occur unless all data is known to be in the current code page. The use of the SUNI attribute is not recommended when 16.2 Developing New Applications with Unicode .

Also See

- 16.1.2 Other Files
- 16.1.3 Intrinsic Functions
- 16.1.4 BIFs and Unicode
- 16.1 Unicode Support in LANSA

16.1.2 Other Files

Load Other Files automatically creates new field definitions as required. If the underlying database column is stored in the database as Unicode, LANSA will automatically create or reuse an appropriate field along the following guidelines:

- If the file definition already exists, and it contains any Char or String fields with the SUNI attribute, any new fields created for Unicode database columns will be created as Char, String, or CLOB fields with the SUNI attribute.
- The only exception to this rule is the IBM i native type Alpha with CCSID 1208 (UTF-8), which is always loaded as Nchar or Nvarchar.
- If it is a new file, or there are no Char or String fields with the SUNI attribute on the file, Load Other File will create Unicode fields as Nchar or Nvarchar.

Refer to 16.3.4 Other Files with SUNI Fields.

for details on how you might change existing Other Files from using SUNI fields to 16.1.1 Unicode Field Types.

Also See

16.2 Developing New Applications with Unicode

16.3 Modifying Existing Applications to use Unicode

Load Other File in the Visual LANSA Developer's Guide or Load Other File in the Technical Reference Guide.

16.1 Unicode Support in LANSA

16.1.3 Intrinsic Functions

LANSA provides Intrinsic functions for use in RDMLX code.

The following functions are particularly useful when developing with 16.1.1 Unicode Field Types:

Intrinsic Function	Notes
AsNativeString	Use this function to convert Nchar or Nvarchar data to the current code page when Unicode Field Types are not supported. For example, when invoking a BIF that doesn't support Unicode. Characters not supported in the current code page are changed to ?.
AsUnicodeString	Use this function to populate an Nchar or Nvarchar from the numeric value (code point) of the character. It converts a decimal whole number to an individual Unicode character – these can be concatenated to construct a Unicode string that may not be possible to represent in the current code page.
AsCodePoint	Use this function to deconstruct an Nchar into its individual characters as numeric values. This function is the opposite of AsUnicodeString. It takes an Nchar character and returns the numeric value (code point) of the character.

Also See

Alphanumeric/String Intrinsic Functions in *Technical Reference Guide* 16.1 Unicode Support in LANSA

16.1.4 BIFs and Unicode

LANSA has implemented strong typing for all field types. For Built In Functions, this means you cannot pass an Nchar or Nvarchar field as an argument or return value unless the documentation indicates Unicode field types are accepted. Refer to Built-in Function Rules in *Technical Reference Guide* for a list of the argument and return value types that can take Unicode field types.

If Unicode field types are not accepted as arguments, they can be coerced to native string format using AsNativeString as mentioned in 16.1.3 Intrinsic Functions. If Unicode field types are not accepted as return values, you must use an acceptable field and then assign the value of this field to the Nchar or Nvarchar field. For example:

```
Use Builtin(UPPERCASE)
With_Args(#MYNCHAR.AsNativeString)
To_Get(#STD_TEXTL)
#MYNCHAR := #STD_TEXTL
```

However, if the text you are working with is not in the current code page, you may simply end up with a string of question marks.

Most similar BIFs can be replaced by an Intrinsic Function with equivalent functionality. Always check for an alternative that will avoid data loss by allowing 16.1.1 Unicode Field Types to be used directly. For example:

#N28001501.UpperCase

TRANSFORM_FILE and TRANSFORM_LIST

The TRANSFORM_FILE and TRANSFORM_LIST BIFs support populating lists from disk files and writing lists to disk files.

These BIFs can support files encoded in UTF-8, and this format is recommended when handling 16.1.1 Unicode Field Types. Refer to the File Format argument for details on how to specify that UTF-8 is used.

Also see

Refer also to Built-in Functions by Category in the Quick Reference in the *Technical Reference Guide*.

 \Uparrow 16.1 Unicode Support in LANSA

16.1.5 Web Application Modules (WAMs)

- WAMs support the display and editing of 16.1.1 Unicode Field Types.
- The generated XML for WAMs is always output in UTF-8.
- Technology Service Stylesheets are encoded for UTF-8.

Also See

16.2 Developing New Applications with Unicode Web Application Module Guide

 $\ensuremath{\Uparrow}$ 16. Internationalization with Unicode

16.1.6 Multilingual Descriptions

- The Visual LANSA IDE allows all languages to be entered on a machine of any language, provided operating system support for that language has been installed.
- All multilingual text in the Visual LANSA IDE is displayed using Unicode. You can see and edit English, French, Japanese, Chinese, etc. exactly as it will appear in the target application when running in a particular language.
- Multilingual text is saved in the repository as UTF-8 converted to hexadecimal.
- ↑ 16.1 Unicode Support in LANSA

16.2 Developing New Applications with Unicode

Always use an RDMLX-enabled partition when developing new applications (or extending existing applications). For further suggestions, refer to:

- 16.2.1 Field types and Language-specific data
- 16.2.2 Types of LANSA Applications

Also See

16.3 Modifying Existing Applications to use Unicode

16.1 Unicode Support in LANSA

16.2.1 Field types and Language-specific data

- For short fields that will be a code in uppercase, continue to use Alpha or Char. For example, ISOCountryCode Alpha(2) can hold the ISO country codes like AU, CA, DE, FR, GB, JP, US.
- For code descriptions, use an Nchar or Nvarchar. For example, CountryDescription Nvarchar(20). If you wish to have language-specific descriptions, store the language with the data when it is entered. This would allow the description to be retrieved by language. For example, create a file called Countries, which has a two-part Primary Key ISOCountryCode, Language. This would allow the appropriate description to be returned for the language I am running in. The following code would set #CountryDescription appropriately for the language, so if LANG=ENG "Germany", if LANG=DEU "Deutschland", if LANG=FRA "Allemagne" and so on.

#ISOCountryCode := DE Fetch Fields(#CountryDescription) From_File(Countries) WITH_KEY(#ISOCountryCode *LANGUAGE)

• For data that may contain unusual characters, use an Nchar or Nvarchar. For example, universities need to print a student's name correctly on certificates, so they might create EnglishName String(50) and ExactStudentName Nvarchar(50). This would allow a file like the following:

StudentID	EnglishName	ExactStudentName	
100953	Mikhail Gorbachev	Михаил Горбачёв	
100954	Bjork Gudmundsdottir	Björk Guðmundsdóttir	
100955	Ang Lee	李安	
100956	Sinead O'Connor	Sinéad O'Connor	

Also See

16.1.1 Unicode Field Types

Field Type Considerations in the Technical Reference Guide

↑ 16.2 Developing New Applications with Unicode

16.2.2 Types of LANSA Applications

- The Universal Model, described in the *Visual LANSA Developer's Guide* is NOT recommended when working with Unicode data. It does not support display of Nchar fields these must be copied to Alpha fields and that means data loss if the data is not all in the current code page.
- The best choices for working with Unicode data because they allow data from different languages to be displayed and edited in the same form or webpage are:
 - the Windows Optimized Model described in the Visual LANSA Developer's Guideand
 - the Web Application Modules described in the *Web Application Modules* (*WAM*) *Guide* are
- The Visual LANSA Framework is an optional extension to LANSA which provides an application framework for designers and developers. A key benefit of the Framework is that from a single application model it supports deployment to Windows Rich Client, Web Browser, or Microsoft .NET.

Also See

16.1 Unicode Support in LANSA

Creating LANSA Applications in the Visual LANSA Developer's Guide Visual LANSA Framework Guide

↑ 16.2 Developing New Applications with Unicode

16.3 Modifying Existing Applications to use Unicode

Is the partition RDMLX enabled?

You can only use 16.1.1 Unicode Field Types in an RDMLX partition. Read Enable Existing Partitions for Full RDMLX in the *Visual LANSA Administrator's Guide* and ensure you understand the ramifications of enabling the partition for RDMLX.

Changing field type from Alpha, Char or String to Nchar or Nvarchar

There is always some risk involved in changing a field type. Risks include:

- Fields are used by external applications (including accessing tables on which the fields appear as columns) that will not know how to handle Nchar or Nvarchar data.
- Fields are used in parts of LANSA that do not support Nchar or Nvarchar. For example, they may be used on RDML functions or as BIF arguments that do not support 16.1.1 Unicode Field Types.
- Missing some objects that refer to the field, and not rebuilding or testing the objects.
- The change yields no benefit as the data is not multilingual. For example code strings in lookup tables.
- Assignment occurs between Unicode and non-Unicode fields necessitating even wider changes than envisaged.

Many of these risks can be reduced or removed by NOT changing the field type, and instead copying existing fields to new fields with the new field types, which can then be mapped to and from the existing fields to preserve existing functionality. This is the recommended approach for Alpha fields used in RDML functions or RDML files. Keep in mind that data loss may occur with mapping between Unicode field types and Alpha, Char, and String unless all data is known to be in the current code page.

If you change a field on a file from Alpha, Char or String to Nchar or Nvarchar, data is preserved correctly when you rebuild and reload the file data (with the IDE or via the Deployment Tool or LOAD_FILE_DATA BIF). If you change a field from Nchar or Nvarchar to Alpha, Char or String, data loss may occur.

Do you need to reload other file definitions?

You may also wish to reload existing Other Files so as to take advantage of the new field types available, especially Nchar and Nvarchar if the files contain Unicode columns.

To reload an RDML Other File as RDMLX, with RDMLX fields where appropriate, the original file definition must be deleted.

To modify an RDMLX Other File that has SUNI fields, so that you can now access those fields via 16.1.1 Unicode Field Types, refer to 16.3.4 Other Files with SUNI Fields.

More Information

For suggestions on how to modify existing applications to take advantage of Unicode, please review the following:

- 16.3.1 RDML Functions
- 16.3.2 LANSA for the Web
- 16.3.3 Windows Optimized applications

Also See

- 16.2.1 Field types and Language-specific data
- 16.2.2 Types of LANSA Applications
- 16. Internationalization with Unicode

16.3.1 RDML Functions

Applications based on RDML functions, the Universal Model described in the *Visual LANSA Developer's Guide*, are the most difficult to convert to Unicode because RDML functions are the oldest kind of LANSA development and do not support new features such as Unicode.

Generally, RDML objects cannot refer to RDMLX objects, however RDML functions can use the EXCHANGE and CALL commands to invoke RDMLX functions to do a unit of work. Note that RDMLX functions do not have a user interface: the DISPLAY, POP_UP, and REQUEST commands are not supported on RDMLX functions except in LANSA for Web, but these do NOT support 16.1.1 Unicode Field Types.

The following information assumes the partition is enabled for RDMLX, but all fields, files and functions are RDML and no Components are used.

Extend or Rewrite?

There are two approaches that can be used:

- Extend the existing application. Leave much of the application as is, and create new RDMLX fields, files, components, functions, and optionally Web Application Modules (WAMs) that utilize existing RDML objects where required. RDML functions can be altered to call RDMLX functions to do Unicode specific processing when no change to the user interface is required. Refer to 16.2 Developing New Applications with Unicode for suggestions.
- Rewrite the existing application, minimizing use of the Universal Model. Decide what your main application style will be: a Windows Optimized Model in the Visual LANSA Developer's Guide or Web Application Modules as described in the Web Application Module (WAM) Guide. Most Alpha fields can be left as is. Change certain Alpha fields to be Char or String (if you need more than 256 bytes). Change other particular Alpha fields to Nchar or Nvarchar if you need to support multiple languages. Add other RDMLX field types as required.

Also See

16.1 Unicode Support in LANSA

Load Other File in the *Visual LANSA Developer's Guide* or Load Other File in the *Technical Reference Guide*.

16.3 Modifying Existing Applications to use Unicode

16.3.2 LANSA for the Web

LANSA for the Web has two main application types:

- Webevent Functions: These are special functions that generate HTML or XML for DISPLAY, REQUEST, and POP_UP commands. They do not allow 16.1.1 Unicode Field Types to be displayed and edited. However, RDMLX Webevent functions do allow other RDMLX field types to be displayed, unlike 16.3.1 RDML Functions.
- 16.1.5 Web Application Modules (WAMs): These are LANSA's solution for building applications that deliver their User Interface in a form of XML, typically via a web browser. WAMs do allow the handling of 16.1.1 Unicode Field Types.

If your application consists mainly of Webevent Functions, the recommended approach is to extend your application with WAMs where Unicode data needs to be displayed. Refer to WAM and Webevent Interoperability in the *Web Applications Modules (WAM) Guide* for recommended techniques.

If your application consists mainly of WAMs, there are no special instructions, just enhance your application as desired.

Also See

- 16.1 Unicode Support in LANSA
- 16.3.1 RDML Functions
- 16.3 Modifying Existing Applications to use Unicode

16.3.3 Windows Optimized applications

If your application consists mainly of Components, there are no special instructions, just enhance your application as desired.

Also See

16.1 Unicode Support in LANSA

16.3 Modifying Existing Applications to use Unicode

16.3.4 Other Files with SUNI Fields

When an existing Other File contains fields which have the SUNI attribute, you may wish to re-load the file and its fields to use Nchar and Nvarchar. To do this, you need to remove all Char and String fields with the SUNI attribute from the file, and then reload the file.

Here's an example of how to do this. You can modify these instructions to suit your own needs:

Prepare for the change

- 1. Check out the file and its fields that have the SUNI attribute.
- 2. Create new fields that are copies of any SUNI fields on the file. These can be deleted later if not required.
- 3. Create a new list that contains the SUNI fields and all objects that use the fields, as well as the new copied fields. Save this list to Excel.
- 4. Make note of any *WEBEVENT functions that use SUNI fields in DISPLAY, REQUEST or POP_UP commands.
- 5. Check out all the objects that use the fields.
- 6. Print the file definition.
- 7. Delete the SUNI fields from the file.
- 8. Delete the SUNI fields from the repository.

Reload file creating new fields

- 1. Reload the file. It should now have Nchar or Nvarchar fields replacing the original SUNI fields.
- 2. Update the new fields with rules and triggers, multilingual descriptions, etc from the original fields: information will have been saved when you copied the original fields earlier.
- 3. If any SUNI fields were used in DISPLAY, REQUEST or POP_UP commands in *WEBEVENT functions, you will need to continue to use SUNI fields here. The simplest technique is to take the copied field and copy it back to the original identifier, and then use this as a virtual field on the file. Remember that data loss may occur when Unicode field types are assigned to SUNI fields unless all data is known to be in the current code page.

Finish the conversion

- 1. There is no guarantee that newly created fields will use the same identifier as the old field. If the identifiers are different, and the field is used outside of the file definition, there are two options:
 - Use the Excel list created earlier to change the objects to refer to the new field identifier instead of the old field identifier.

or

- Copy the field created by Load Other File to a new field with the old field identifier. Then use these as virtual fields on the file.
- 2. Rebuild the OAM and the objects that use the field, and re-test functionality.
- 3. Check everything back in.

Also See

- 16.1.2 Other Files
- 16.1.1 Unicode Field Types
- 16.3 Modifying Existing Applications to use Unicode

•

17. Client/Server Applications

Following is an introduction to some of the basic client/server models that are supported by Visual LANSA applications.

They have been written to provide an understanding of the types of client/server models that you may choose to work with.

They are not written to provide technical or specific information.

- 17.1 The Basic Client Model
- 17.2 Basic Client/Server Models
- 17.3 Using ODBC in a Client/Server Model
- 17.4 Using LANSA SuperServer
- 17.5 Designing Client/Server Applications
- 17.6 Client/Server Applications Myths, Legends and Expectations
- **17.7 Performance Issues**

17.1 The Basic Client Model

In terms of database access the most simple (and most common) Visual LANSA generated application works like this:



17.2 Basic Client/Server Models

When you are using ODBC, changes made to the database or data source configuration usually allows the previous application to act in a client/server mode like this:

Windows PC



17.3 Using ODBC in a Client/Server Model

The previous client/server model under Windows can be taken outside the Windows realm by having the correct ODBC drivers available for your chosen DBMS. For example:



As you can see, it goes from one part of the DBMS to another part of the DBMS.

The Visual LANSA application has nothing at all to do with this communication between parts of the DBMS.

Sometimes people ask what communication protocols do Visual LANSA applications support? This question often arises in situations where database vendors (such as Oracle, Sybase, etc.) are involved.

From the diagram, it can be seen that Visual LANSA applications are generated to be "generic" database users.

Visual LANSA applications are designed to use other vendor's DBMS products like Sybase, Microsoft, Oracle, etc.

Visual LANSA is not, and does not contain, a DBMS system.

So the question of what communication protocols Visual LANSA can use in client/server environments should be directed to your DBMS vendor.

17.4 Using LANSA SuperServer

Finally there is the LANSA SuperServer client/server model that the LANSA supports.



Since LANSA for iSeries can create OAMs (IO Modules) functionally identical to those it creates under Visual LANSA, this special or "enhanced" client/server model can be supported.

Note: LANSA SuperServer can also be used to access databases on a Windows platform. The benefits described here for IBM i also apply to those platforms. Some of the more important facilities that this model provides are:

• Application Security

All access requests are subjected to the same set of authority checks as normal IBM i applications.
- Application Integrity All access requests are subjected to the same set of validation rules as normal IBM i applications.
- Performance This client/server model offers good client/server performance for two main reasons:
- It is specifically optimized for IBM i processing.
- The OAM (IO Module) architecture complements the LANSA Data Dictionary. e.g. When an "INSERT" request requires several database validations and fires an event trigger it is all initiated by just one simple Client request. The level of traffic between the client and server is minimized.
- Multiple Connections This model allows an application to connect to multiple IBM i databases simultaneously.
- Simplicity and Cost

There is no extra software or cost for each PC accessing an IBM i server this way. The IBM i server needs a single "LXX" (LANSA SuperServer) license. There are no extra install procedures and the connections are all via standard Windows based communication routers.

- Compatibility and Reversion This model is fully supportive of, and complementary to, the DDCS/2 DRDA/ODBC models. In fact you can, at any time, drop this model and revert to the DBRA/ODBC model without any need to change your application design.
- Remote Procedure Calls (RPCs)

This model has a Built-In Function named CALL_SERVER_FUNCTION that allows you to call a function on the server. Parameters and lists can be passed to and from the server function.

The presence of an RPC capability is probably the most important component of a client/server model. You will find that you cannot possibly create an application system where all components run on the client system CPU(s). There will be many activities that have to be performed on the server.

The LANSA SuperServer client/server model is supported by these Visual LANSA Built-In Functions:

DEFINE_OS_400_SERVER	Define an IBM i server machine.
DEFINE_OTHER-SERVER	Define other server.
CONNECT_SERVER	Connect to a server.
DISCONNECT_SERVER	Disconnect from a server.
CONNECT_FILE	Connect a file to a server.
DISCONNECT_FILE	Disconnect a file from a server.

CALL_SERVER_FUNCTION Call a function on a server.

For full technical details refer to *Built-In Functions* in the *LANSA Technical Reference Guide*.

17.5 Designing Client/Server Applications

The word "design" here does not simply refer to the design of the User Interface with its action bars, push buttons, drop downs, etc. but to much larger and infinitely more complex issues that you will have to resolve as part of the design process.

Some of the things that you may need to think about include:

- Designing a Distributed Database This brings a new level of complexity to applications. What's on the client(s)? What's on the server(s)? How much disk space? What client/server model is to be used? Object Locking?
- Designing a Distributed Application What runs on the client(s)? What runs on the server(s)? What runs on both? What client/server model is to be used?
- Performance and Communications Loads How much load on the LAN? Can it run on twinax? What about slow leased line and dial-up access?
- Security and Integrity Considerations Backup what? Backup when? Backup where? Software distribution and upgrade procedures across "n" PCs? Security administration across "n" PCs?
- The User Interface What about OOD? GUI WIMP constructs? Action bar object-action designs? This is not a traditional IBM i "menu driven" system.

The golden rule here is, **DO A PILOT STUDY**.

Invest enough time to gain experience in designing client/server applications by doing a small to medium sized pilot project. Do not leap into a large scale, mission critical project first off. Give your designers time to study and learn new techniques and gain critical design experience before embarking on a large-scale project.

In summary, some client/server design dos and don'ts:

DON'T

- Start with a critical application.
- Design traditional NPT applications and try to run them on the client with all database access to the server database.
- Ignore resource loads until the end of the project.

DO

- Spend a lot of time on design.
- Do not design for NPTs (non-programmable terminals).
- Distribute data according to whether INQUIRE or TRANSACTION intensive.
- Estimate & test & validate that the resource loads can be handled before undertaking the project.

Data validation:



Selecting records from multiple files on the server:



Selecting a high volume of records on the server for calculations:





Passes results back to the Client

17.6 Client/Server Applications - Myths, Legends and Expectations

While the concept of client/server applications is no longer new, there is still a high level of unreal expectation from these applications. To avoid becoming a failed client/server statistic, it is recommended that you always keep, in the back of your mind, 3 of the greater myths of the 20th century:

- The cheque/check is in the mail.
- You can build a really effective and fast client/server based EIS over your existing 15 year old database simply by buying a generic EIS tool.
- That twinax and LAN attached communications subsystems have net data throughput rates close to that of channel/bus attached hard disk drives.

The last two points need some elaboration:

EIS (Executive Information Systems)

Everybody has seen the demonstration. The executive clicks on an icon and 2 seconds later a pan-dimensional chart of product sales, grouped by category, is shown, year to date, for the last 25 years. The graph is then used at the next board meeting and the executive receives much acclaim.

While every MIS department would like to provide this type of facility to their executives, without understanding and planning for some of the basic EIS requirements it will remain exactly what it is, a fairy story.

Unless you have planned, architected and designed EIS information into your database you will not be able to create results like this within reasonable time frames.

In other words, the previously mentioned EIS scenario may well be possible, but if you keep 300,000 rows of product line item information it may well take 90 minutes to extract and summarize the information required for the pandimensional graph. And, even worse, if the executive does not know the significance of the "Record Deleted (Y/N)" column in the database table, all he/she may receive at the next board meeting is severe embarrassment.

Some things you may care to think about:

- EIS systems need to be planned and carefully designed.
- EIS means summaries.
- Summaries mean reading lots of rows of data.
- Reading lots of rows takes time and consumes a lot of CPU power.

- Most useful and rapid EIS systems keep "pre-summarized" data.
- The end users actually query the summary data.
- The summary data may be updated dynamically or cyclically.
- Designing and maintaining summary data means extra design and programming effort independent of any generic tool being used.

Data throughput

This is really an extension of the previous point.

Don't go into client/server with unrealistic expectations.

You may design an application that "sucks up" 6,000 table rows into a spread sheet on your PC and then summarizes and sorts them.

However, you should then ask yourself some questions like these:

- How long will it take across a twinax connection?
- What will happen to the other twinax users while this is happening?
- What will happen to my server if 40 users try to do this at the same time?
- What will happen if one user requests this every 30 seconds?
- Would using an RPC (Remote Procedure Call) to summarize the 6,000 rows on the server and then sending the summarized results back to the client be a better solution?
- Should some sort of "drill down" approach be used? Maybe then you would only have to "suck up" 100 records? How can the "drill down" approach be used with this application?

There seems to be some sort of belief in some sections of the market that client/server applications work by magic. For instance:

- Many of the old DP rules about benchmarking and load testing, etc. seem to have been lost in the scramble for client/server applications.
- **The old rules still do apply**. If you ignore them until your application is complete then you are taking a large risk.
- Building client/server applications and using client/server tools may place a great deal more load on your LAN and twinax communication subsystems. Can they handle this increased load?

To Summarize:

- Spend a lot of time in design
- Use virtual fields to do dynamic "joins"

- Use triggers
- Put validation rules into the data dictionary and I/O Modules
- Distribute load/logic across the Client and Server
- Avoid pushing too much data through communication subsystems
- Do realistic tests/benchmarks before implementation
- Note that the amount of data transferred includes not just data from the database on the Server to the Client program but also from the Client program to the UIM (User Interface Manager).

This is very important and often overlooked.

17.7 Performance Issues

Testing and Balancing Across Multiple Platforms

Different operating systems are good at doing different things.

For example, IBM i is very good at doing database I/Os and Windows is very good at CPU intensive operations like SORT_LISTs.

You **cannot** sensibly estimate what an applications performance will be like on one platform by testing the application on another platform.

It is your responsibility to find the right balance between differing hardware platforms. Your application design must consider how the platforms are different and determine how to best utilize the specific features the platform has to offer.

Amount of Information in Panels/Dialogs/Windows

Do not overload screen panels, dialogs or Windows.

For example a screen panel containing 100 fields may be slower to present when executing under Windows compared to IBM i.

"Drawing" time in advanced graphical environments is directly related to the amount of information (i.e. number of objects) in the windows.

PC, Desktop and LAN Based DBMS

Generally IBM i developers form their perceptions about the speed and methods by which they can access a database table from their IBM i experiences. Unfortunately this may be the worst possible place to gain these perceptions.

The IBM i is a multi-user database machine. Its database access is very fast and relatively little attention has to be paid to the impact of DBMS requests on the system or other users.

However, as we move more and more towards PC, Desktop and LAN based DBMS, developers may have to alter their perceptions and change their habits accordingly.

This is a subjective area, and PC DBMS vendors may dispute this, but development experiences so far have indicated:

- Rarely will a PC based DBMS perform as well as an IBM i DBMS.
- Some PC based DBMS are really only good for the home user, even though they may be sold as being suitable for high volume, multi- user commercial use.

• Most developers test their applications with very small data sets and will not identify a DBMS performance problem area until after the application gets into the hands of production users.

So, in any context where you are using a PC based DBMS, please think about the following before you implement an application:

- There are hardware considerations. For example, an older Pentium PC using an IDE hard drive may have "useable" disk access times 5 times slower than a newer Pentium PC with a SCSI drive / controller / cache.
- Avoid SX computers. When decimal numbers and high volumes are involved make sure that you are using a DX computer that has a floating point co-processor. Cases we have seen indicate that in numerically intensive high volume database access a DX computer may be 4 to 6 times faster than an equivalent SX.
- Think about loads and volumes during the design phase.
- The most likely area of problems is to do with tables that contain large numbers of rows and SELECT commands. Slow SELECT's are the most common cause of PC DBMS problems.
- Be CAREFUL when using SELECT on a table that you know will have a large number of rows. If there is no index to support your SELECT, then the DBMS will read the whole table. If there are 100,000 rows, then this operation WILL take a long time. Even if there appears to be an index, the DBMS may decide to read the whole table anyway.

However, if you are testing with realistic volumes then you can identify and remove the problem area before it gets to production users.

- Be RESPONSIBLE. It is no good designing and implementing an application without any thought for load, volumes and viability and THEN finding it is too slow to be viable, and then complaining to LANSA or to the DBMS vendor. You need to AVOID the situation in the first place.
- SELECT_SQL is the fastest type of DBMS access. It creates imbedded SQL in the generated C code. This is the fastest type of access that can be done. If you can't achieve a viable result using SELECT_SQL then you will have to find an alternative solution to the problem.
- When very large tables are involved, consider placing them on an IBM i server and then using LANSA SuperServer or Remote Function calls to access them, possibly only returning the summarized result(s) in a working list.

• Prototype and test under realistic loads and volumes any table that is expected to have more than 1000 rows in it. This way you should find, and remove, any problem areas before they become critical.

The 1000 figure is totally arbitrary. From your own experiences with some PC based DBMS you may choose to lower this figure to as low as 100 for some PC DBMS, and for others, raise it up to 5000.

The performance of PC based DBMS covers a very broad spectrum. After you have chosen one, you will have to develop a perception of what its capabilities are and thus at what point you have to start being careful.

• Become familiar with your selected DBMS. Read the sections in any guides / manuals that it provides related to performance and tuning. Learn to optimize its performance.

Disk Space Considerations

When using Visual LANSA to develop and test IBM i based applications, you should not keep source or objects that are not required to execute the application (e.g. .C, .H, .STG, .DEF and .MAK files) on your PC. By removing these files, you will very substantially increase the amount of available disk space on your PC.

If you are creating stand-alone applications for Visual LANSA, you should also consider compiling the final version of your programs without debug enabled. Compiling without debug will further increase available disk space.

18. Icons and Bitmaps

- 18.1 The Importance of Standards and Guidelines for Icons and Bitmaps
- 18.2 Using Icons and Bitmaps on a Process Menu
- 18.3 How Icons and Bitmaps are Associated on Process Menus
- 18.4 Using Icons on a Toolbar
- 18.5 How Icons Are Associated and Presented on Toolbars
- 18.6 The X_ICOBMP.DAT Defaulting File
- 18.7 Frequently Asked Questions about Icons and Bitmaps

18.1 The Importance of Standards and Guidelines for Icons and Bitmaps

The ability to put icons and bitmaps onto presentation spaces is an important feature of GUI design.

However, if you do not have standards and guidelines for their usage, they may become a "double edged sword".

If you leave the choice and use of bitmaps and icons to individual programmers you may rapidly develop a quite complex mess.

Many different icons and bitmaps will be used for the same thing.

There will be little re-use of existing icons and bitmaps.

The entire idea of using icons as a standardized visual prompt for common activities may become totally lost in a proliferation of hundreds of overlapping, subjective and confusing icons.

For example, pre-CUA, there was no standard for the "Exit" function key. This lead to many confusing ways and methods by which you could exit from an NPT application.

If you use icons and bitmaps with no standards or guidelines, you may end up in a similar position - with 10 different "Exit" icons, all of which will confuse your end users, lower their productivity and increase their learning curves.

It is strongly recommended that you:

- Use and enforce system wide standards for icon and bitmap usage. Do not leave this job to individual developers.
- Assign someone as the "icon and bitmap administrator".

The following pages describe how icons and bitmaps are implemented within Visual LANSA. This method of implementation was deliberately designed to remove and isolate the icon and bitmap choices from the arbitrary programmer decisions and to assist in the implementation of consistent and standardized bitmap and icon usage across all your applications.

18.2 Using Icons and Bitmaps on a Process Menu

If a Visual LANSA process menu (i.e. the Process Style is "MENU") is invoked, it will be presented something like this:

BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII <text description=""></text>
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII <text description=""></text>
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII <text description=""></text>
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII <text description=""></text>
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	IIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	

The area represented by the "BBBBBB"s is a bitmap that has been associated with the process, and the "IIII"s represent icons that have been associated with the options that may appear on the menu.

This section describes how bitmaps and icons are associated with processes and functions. Once this technique is understood, complete control of what bitmaps and icons appear on menus can be achieved.

18.3 How Icons and Bitmaps are Associated on Process Menus

When a MENU style process is compiled, a standard method is used to decide what icons and/or bitmaps should be associated with itself and any object it references (i.e. the associated RDML functions, menu commands, attached process or functions, etc.,).

The association method (which is performed and locked in at compile time) works like this:

- 1. The "object" name (process or function) is converted into the standard name format. Usually this is identical to the object name unless it contains special characters or is too long. The method used for this conversion process is described in earlier sections of this guide.
- 2. A check is made in the current partitions SOURCE directory (i.e. <drive>:\X_LANSA\X_ppp\SOURCE) for a file with the same name as the object, suffixed by .ICO (icon) or .BMP (bitmap).
- 3. If such a file exists, the association is complete and none of the following steps are used

If step 2 does not find an association, a more complex step is invoked. This step uses a file called X_ICOBMP.DAT that may exist in the current partitions SOURCE directory (i.e. <drive>:\X_LANSA\X_ppp\SOURCE). This file is called the icon and bitmap defaulting file.

The contents of file X_ICOBMP.DAT are used to try to produce an "intelligent" default icon or bitmap association. This is done by comparing data in X_ICOBMP.DAT with name and textual information associated with the current object.

The exact layout and use of X_ICOBMP.DAT is described in detail in a following section.

If step 3 does not find an association, a "final defaulting" step is used. This produces final defaults according to this table:

Menu Object	Default Icon And/Or Bitmap Used	
Process	X_DFTPRO.ICO	X_DFTPRO.BMP
Function	X_DFTFUN.ICO	
Menu Command	l X_DFTCMD.ICO	
Help	X_DFTHLP.ICO	

Return to	X_DFTRET.ICO
Exit	X_DFTEXT.ICO
all others	X_DFTMIS.ICO

18.4 Using Icons on a Toolbar

If you are using a Visual LANSA action bar (i.e. the Process Style is "ACT/BAR") process, it will be presented something like this representation:

ttttttt ttttttt ttttttt ttttttt

IIIII IIIII

The area represented by the "tttttt"s is the text associated with the action bar items. From each of these individual drop down menus can be displayed/selected.

The "IIII"s represent icons that have been associated with drop down menu options that have an associated icon.

The "BBBBBBBBBB"s represents a bitmap that has been associated with the process (the method of associating the bitmap with the process is identical to that described in the previous section for "MENU" style processes).

Selecting a tool bar item from the tool bar area is functionally equivalent to selecting the same item/function from the drop down menu associated with the action bar.

18.5 How Icons Are Associated and Presented on Toolbars

When an ACT/BAR style process is compiled a standard method is used to decide which functions (i.e. drop down menu items) should also have their associated icon appear in the tool bar.

The association method (which is performed and locked in at compile time) works like this:

- 1. The "object" name (process or function) is converted into the standard name format. Usually this is identical to the object name unless it contains special characters or is too long. The method used for this conversion process is described in earlier sections of this guide.
- 2. A check is made in the current partitions SOURCE directory (i.e. <drive>:\X_LANSA\X_ppp\SOURCE) for a file with the same name as the object, suffixed by .ICO (icon).
- 3. If such a file exists, the association is complete and none of the following steps are used. The function will appear in the tool bar area.

If step 2 does not find an association, a more complex step is invoked. This step uses a file called X_ICOBMP.DAT that may exist in the current partitions SOURCE directory (i.e. <drive>:\X_LANSA\X_ppp\SOURCE). This file is called the icon and bitmap defaulting file.

The contents of file X_ICOBMP.DAT are used to try to produce an "intelligent" default icon by comparing the PD\$OPT (Pull Down Option) value associated with the function (in its current pull down context) to all icons in the default file that use the PDO= option keyword. If a match is found, then the function is selected to appear in the tool bar area and no further checking is done.

The exact layout and use of X_ICOBMP.DAT is described in detail in a following section.

If step 3 does not find an association, then the function will not appear on the toolbar. The function can only be invoked by selecting it from the pull down in the normal manner.

This ability to select which functions appear in the tool bar and the drop down menus, and which only appear in the drop down menus, is quite important.

For example, an action bar may have 7 items, and each item has an associated pulldown averaging 5 items. This means that there are 35 "actions" selectable from the pull down menus.

If all of these had their own icon, then Visual LANSA would try to fit 35 icons on the tool bar. In fact they would not all fit, and many of them would not actually be visible or selectable from the tool bar.

So of the 35, you should choose 5 or 6 that are heavily used (often these are the ones that have associated accelerator keys) and create icons or automatic associations for them. This means that the heavily used 5 or 6 functions are accessible from the pulldown menus and the tools bar, but the less heavily used 30 other functions are only accessible from the pull down menus.

It is also important to note that step 3 allows multiple icons to be associated with a single function.

For example, a single RDML function called FUNC01 may actually "service" 3 pull down options called OPN, CLS and VEW. By using the icon association file you can cause 3 different icons to appear on the tool bar, even if they all cause the same function to be invoked.

18.6 The X_ICOBMP.DAT Defaulting File

The X_ICOBMP.DAT icon and bitmap defaulting file may optionally reside in the partition SOURCE directory (i.e. <drive>:\X_LANSA\X_ppp\SOURCE where "ppp" is the partition identifier.

It is a standard text file and may be edited with most common source editors such as Notepad.

Each individual line should start with one of these "keywords" to indicate what the line contains:

ICO Identifies an icon file (without the .ico suffix)

BMP Identifies a bitmap file (without the .bmp suffix)

NAM Identifies an associated object name

STR Identifies an associated search string

PDO Identifies a pull down option code (pd\$opt value)

All keywords, file and object names are automatically converted to uppercase before being used. Note that strings (STR=) are not converted to uppercase and are always compared in the case specified.

If the specified icon or bitmap file cannot be found in the SOURCE directory of the current partition then a warning is issued and the line (and all associated NAM= and STR= options) are ignored.

The use of this file is best indicated by example:

```
ico=cust
nam=LXRSF01
nam=LXRSF02
str=Customer
str=customer
* <----- This line does not start with a keyword
ico=printer so it will be ignored.
str=Print
str=Print
str=print
str=report
str=report
pdo=REP
pdo=PRT
```

First an icon called CUST is identified.

Any process or function named LXRSF01 or LXRSF02 is to be automatically associated with it.

This association could also be achieved by copying the CUST icon to create icons called LXRSF01.ICO and LXRSF02.ICO. However, using this file means that many processes and functions can be associated with a single icon.

Additionally, any object that has textual identification containing "Customer" or "customer" is to be automatically associated with this icon.

Next, another icon called PRINTER is identified.

Any object that has textual identification containing "Print", "print", "Report" or "report" should be automatically associated with it.

Any pull down menu entry that has a PD\$OPT value of REP or PRT should appear in the tool bar area and use the PRINTER icon to identify it.

Some final points:

- No input line should be longer than 50 characters.
- Up to 40 characters of STR= value may be specified.
- STR= values are case sensitive.
- ICO=, BMP=, NAM= and PDO= values are not case sensitive.
- Blank characters following STR= values are stripped and ignored.
- Blank characters preceding STR=values are significant.
- Association priority is from the start of the file to the end. Once a match is found processing stops and no further comparisons are performed.
- A very large number of entries in this file could degrade process compile times. The "icon association" stage of a process compile is indicated by messages. If the time between messages becomes very long it may be because there are an unreasonably large number of entries in this file.
- A set of demonstration icons and a demonstration version of X_ICOBMP.DAT is shipped with Visual LANSA. They are automatically installed into all partitions that exist at the time.

The demonstration icons all start with S_ and have English language associations in the shipped X_ICOBMP.DAT file.

18.7 Frequently Asked Questions about Icons and Bitmaps

Where must icons and bitmaps be placed?

In the SOURCE directory of the associated partition. For example, if the partition was called TST, then the icons and bitmaps should be placed in the <drive>:\X_LANSA\X_TST\SOURCE directory.

What size should I make my icons?

Usually icons are 32 x 32 pixels.

What size should I make my bitmaps?

256 x 256 pixels is the recommended maximum size for bitmaps that are presented on process menus. When the process presents itself as an action bar larger bitmaps may be used.

How many colors should I use in my icons and bitmaps?

Usually you should use 16 colors only.

How do I edit bitmaps under Windows?

The Windows "PaintBrush" facility can easily work with bitmaps of the recommended size, as can the IMAGEDIT toolkit product.

Should certain names be avoided for my bitmaps and icons?

Yes. Avoid using the prefixes X_, S_ or W_ for bitmaps or icons that you create. During a Visual LANSA install or upgrade all X_, S_ and W_ icons and bitmaps may be replaced. This could cause you to lose any changes that you make.

Where does the defaulting file X_ICOBMP.DAT reside?

In the SOURCE directory of the associated partition. For example, if the partition was called SYS, then the icons and bitmaps should be placed in the <drive>:\X_LANSA\X_SYS\SOURCE directory.

Do I have to have an X_ICOBMP.DAT defaulting file?

No. It is an optional file. Process compiles will simply issue a note indicating that the file could not be found.

What language is the defaulting file X_ICOBMP.DAT shipped in? English.

Is the defaulting file X_ICOBMP.DAT replaced when I upgrade?

Yes. This means that every time you alter this file you should back it up. After each upgrade of your Visual LANSA system simply replace the upgraded file with your own version.

When I edit an icon it does not change on the process menu. Why?

Icons and bitmaps are actually "imbedded" into the process DLL during the compile process. If you change an icon or bitmap you must force a recompile of process menus that reference it.

This also means that when you move your application to another machine you do not have to ship the icons and bitmaps as well.

They are only required on the target machine if you intend to perform process (re)compilations there.

Can I alter the X_DFTxxx icons and bitmaps?

Yes. Remember that every time you alter one of these files you should back it up. After each upgrade of your Visual LANSA system simply replace the upgraded file with your own modified version.

19. Create Your Own Built-In Functions

This chapter describes the processes and rules you should follow when developing your own Built-In Functions for IBM i, Windows or Linux platforms. Due to the platform differences when creating User-Defined 3GL Built-In Functions on Windows/Linux and IBM i platforms, these have been described separately in these sections:

- 19.1 Create your own 3GL Built-In Functions on IBM i for RDML Functions
- 19.2 Create Your Own 3GL Built-In Functions on Windows/Linux/IBM i

The instructions for creating your own RDML Built-In Functions for all platforms are in this single section:

• 19.3 Create Your Own RDML Built-In Functions (all platforms)

Also see

Built-In Functions in the Technical Reference Guide.

19.1 Create your own 3GL Built-In Functions on IBM i for RDML Functions

When creating 3GL Built-In Functions for your own use or for distribution to other LANSA sites, there are some rules and conventions that must be followed.

You must ensure that you check your Built-In Functions against the provided checklist to prevent unexpected application failures or portability problems.

19.1.1 3GL Parameters

19.1.2 Naming Conventions for 3GL BIF on IBM i

19.1.3 Steps to Create 3GL Built-In Functions on IBM ii

19.1.4 Example - Define a Built-In Function as a 3GL Program

19.1.5 Example - Define a Built-In Function as a 3GL Subroutine

- 19.1.6 Package 3GL Built-In Functions for Distribution
- 19.1.7 Install Other Vendor 3GL Built-In Functions

19.1.1 3GL Parameters

Fixed Parameters Arguments / Return Value Parameters Alpha or Numeric Parameter Example List Parameter Example

Fixed Parameters

The first three parameters of the program are fixed. These are data structures that are passed throughout the LANSA generated applications:

DC@IDS - Information data structure

DC@EDS - Arrays defined in Information data structure

PR@IDS - Process Information data structure

The layouts of these data structures (as RPG /COPY members) can be found in members DC@ISPEC, DC@ESPEC, PR@ISPEC and PR@ESPEC in file DC@F28 in the LANSA data library.

You must **not** vary the content of any of the storage locations defined by these data structures. Any attempt to do this may violate system integrity and produce unpredictable results.

Arguments / Return Value Parameters

BIF arguments and return values defined in file DC@F48 are set up as parameters for the BIF program or subroutine.

Each of these parameters is passed with accompanying parameter(s) (i.e. every entry in file DC@F48 actually causes two (2) or four (4) parameters to be set up and passed to the Built-In Function program or subroutine).

Alpha or Numeric Parameter Example

For an argument or return value defined in file DC@F48 as type A (alpha) or N (numeric), two (2) parameters are set up/passed to the Built-In Function:

- The first is a 4 character descriptor. It is formatted "llld" where "lll" is the length and "d" is the number of decimals in the argument / return value specified in the LANSA USE command. This value is often useful because it allows the BIF to determine the actual length of the argument / return value in the LANSA USE command.
- The second is the actual argument or return value. Its length and type

matches the maximum length definitions in file DC@F48. Type N (numeric) variables are always passed in packed decimal format. For example:

	-	
С	*ENTRY PLIS	ST
С	PARM	DC@IDS
С	PARM	DC@EDS
С	PARM	PR@IDS
С	PARM	B\$555A 4 < Descriptor
С	PARM	B@555A 3
С	PARM	B\$555B 4 < Descriptor
С	PARM	B@555B 10
С	PARM	B\$555C 4 < Descriptor
С	PARM	B@555C 1

List Parameter Example

For an argument or return value defined in file DC@F48 as type 'L' (list), four (4) parameters are set up/passed to the BIF:

- The first is a decimal 7,0 field. This field contains the maximum number of entries allowed in the list. The value is set from the list definition in the RDML function that uses this Built-In Function.
- The second is a decimal 7,0 field. This field contains the current count of entries actually in the list. This value is set by RDML function. It comes from the number of entries that are actually used by the function before invoking this Built-In Function.
- The third is a decimal 7,0 field. This field points to a "current" entry in the list. Normally this entry has no meaning within a Built-In Function and it should not be referenced or altered.
- The fourth is the actual list. This field is a multiple occurrence data structure (in RPG terms). This field is the data passed by, or about to be passed to, an RDML function as a working list.

For example:

С	*ENTRY PLIS	Т
С	PARM	DC@IDS
С	PARM	DC@EDS
С	PARM	PR@IDS
С	PARM	LXnnnB 70 < Maximum Entries
С	PARM	LCnnnb 70 < Total Entries

С	PARM	LPnnnb 70 < Current Entry
С	PARM	LLnnnB < Actual List

19.1.2 Naming Conventions for 3GL BIF on IBM i

Built-In Function Definition (BIF Name)

- BIF names can be up to 18 characters in length.
- The BIF name is composed of letters of the English alphabet (A through to Z), 0 through to 9 and underscore only.
- The BIF name should be in the format XX_YYYYYYYYYYYYY where

 $\boldsymbol{X}\boldsymbol{X}$ must be either $\boldsymbol{U}\boldsymbol{D}$ for User-Defined or $\boldsymbol{O}\boldsymbol{V}$ for Other Vendor

YYYYYYYYYYYYYYY is the name that will describe, even to the non-programmer, what functions the BIF performs.

Program name

Program names can be up to 8 characters in length.

•

• The program name must be in the format XX@y999 where

XX identifies this as Built-In Function code; i.e. OV for Other Vendor or UD for User-Defined.

y is the type of object, either P for Program or D for Display file

999 is the unique identifier in the range:400 to 600 for User-Defined (UD) BIFs.600 to 999 for Other Vendor (OV) BIFs.

Subroutines

- Subroutine names can be up to 6 characters in length
- The name should be in the format IB@nnn where

IB identifies this as Built-In Function code

nnn is the unique identifier.

• The subroutine member name should be in the format BIFnnnxSPC

where

BIF identifies this as Built-In Function code
nnn is the unique identifier
x is the type
SPC is a required literal
Valid values for x (type) are:
Valid values for x (type) are:
F File specifications (F specs)
Array specifications (E specs)
R Rename specifications (I specs)
Jata structure specifications (I specs)
Subroutine specifications (C specs)
Output specifications (O specs)

A Compile time array data

19.1.3 Steps to Create 3GL Built-In Functions on IBM ii

- 1. Design the Built-In Function.
- 2. Allocate the next unique number. If this is a User Defined (UD) Built-In Function, use the next available number between 401 and 600. If this is an Other Vendor (OV) Built-In Function, then you must use the next available number in the LANSA allocated range. To obtain the next number, send a request to lansasupport@LANSA.com.au
- 3. Give the Built-In Function a name. Use something that even a nonprogrammer would understand. Prefix your own User Defined BIFs with 'UD_' and Other Vendor BIFs with 'OV_' for easy identification.
- 4. Give the Built-In Function program/subroutine a name. Refer to 19.1.2 Naming Conventions for 3GL BIF on IBM i for guidance if necessary.
- 5. Complete a Built-In Function definition form. You can print the example supplied in 19.4 Built-In Function Sample Form or one that you have designed yourself.

Note: This step is not compulsory but it is a good idea if this is your first Built-In Function. It is meant as an aid to translating your Built-In Function design into the necessary code for use with LANSA.

6. Enter the relevant information into the Built-In Function definition files (DC@F47 and DC@F48). This can be done by creating DFU applications over files DC@F47 and DC@F48, or by writing a simple data entry program over these files. If this data is incorrect, then the Built-In Function will not work as expected.

Note: This step must be performed. If the Built-In Function is not described in these files then it will not be recognized by LANSA.

7. Code your program or subroutine. Use the sample 19.5 Built-In Function Skeletons as a guide to coding Built-In Function programs/subroutines.

19.1.4 Example - Define a Built-In Function as a 3GL Program

This section goes through the steps involved in creating a Built-In Function as a program. The example shown is very simple, but should give an insight to how a Built-In Function is plugged into a LANSA system.

SCENARIO: This is a User Defined Built-In Function. The Built-In Function is used to retrieve system values.

A simple CL program. The program will be passed the system value name and will return the system value.

The name will be UD_GET_SYSTEM_VALUE.

The next available identifier at this site is 411.

BIF Name:	UD_GET_SYSTEM_VALUE
Unique Identifier:	411
Description:	Retrieve a system value
Call or Execute:	C (C or E)
Program Name:	UD@P411
Terminates between calls:	N (Y or N)
Number of arguments:	1
Number of return values:	1

BIF Definition (as per DC@F47 file)

BIF Arguments (as per DC@F48 file)

BIF Name:	UD_GET_SYSTEM_VALUE
Parameter Type	ARG
Parameter Sequence:	1
Parameter Number:	1
Parameter Identifier:	А

Description:	Na	me of system value
Required / Optional:	R	(R or O)
Parameter Type:	А	(A, N or L)
Minimum Length:	1	
Maximum Length:	10	
Minimum Decimal:		
Maximum Decimal:		
Pass Length:	10	
Pass Decimal:		
Default:		

BIF Return Values (as per DC@F48 file)

BIF Name:	UD_GET_SYSTEM_VALUE
Parameter Type	RET
Parameter Sequence:	1
Parameter Number:	2
Parameter Identifier:	В
Description:	Value of system value
Required / Optional:	R (R or O)
Parameter Type:	A (A, N or L)
Minimum Length:	1
Maximum Length:	256
Minimum Decimal:	
Maximum Decimal:	
Pass Length:	256

Pass Decimal:	

Enter the data into the Built-In Function definition files:



UD@P411: PGM PARM(&DC@IDS &DC@EDS &PR@IDS &B\$411A &B &B@411B)

/*=====================================	
/* Variable declarations	*/
/*	
DCL &DC@IDS *CHAR 1	024
DCL &DC@EDS *CHAR 1	.024
DCL &PR@IDS *CHAR 10)24
DCL &B\$411A *CHAR 4	
DCL &B@411A *CHAR 1	10
DCL &B\$411B *CHAR 4	
DCL &B@411B *CHAR 2	56
DCL &\$PGMNM *CHAR	10
DCL &PR@PGM *CHAR	10
DCL &PR@B@N *CHAR	3
DCL &PR@B@A *CHAR	1
DCL &DC@RET *CHAR	1
DCL &DC@MID *CHAR	7
DCL &DC@MVR *CHAR	132
/*=====================================	=======================================
/* Global error handler	*/
/*=====================================	

MONMSG MSGID(CPF0000 MCH0000) EXEC(GOTO ARGERR)

/*	
/* Program mainline	*/
/*	

/* Set up some fields and map some from the data structures */

CHGVAR	&\$PGMNM	'UD@P411'
CHGVAR	&PR@PGM	%SST(&PR@IDS 424 10)
CHGVAR	&PR@B@N	%SST(&PR@IDS 157 3)
CHGVAR	&PR@B@A	%SST(&PR@IDS 160 1)

/* Initially assume that a "good" return will be the result */

CHGVAR &DC@RET 'Y'

/* Perform the required evaluation / action */ /* Retrieve the system value */ RTVSYSVAL SYSVAL(&B@411A) RTNVAR(&B@411B) MONMSG (CPF0000 MCH0000) /* End of program processing logic */ ENDPGM: CHGVAR %SST(&DC@EDS 38 1) &DC@RET CHGVAR %SST(&DC@EDS 39 7) &DC@MID CHGVAR %SST(&DC@EDS 46 132) &DC@MVR RETURN /* ARGERR :Handle a detected error in argument(s) passed to program */ Caller should set DC@MID and DC@MVR to reflect the /* */ /* cause of the error before executing this logic. Note that */ the entire program terminates when this logic is invoked. */ /* It will cause the calling RDML program to fail and issue */ /* /* the message details returned to it in DC@MID/DC@MVR */ _____

ARGERR:

/* Route messages any additional messages back to the caller */

CALL PGM(DC@P9007) PARM(&\$PGMNM &PR@PGM) MONMSG (CPF0000 MCH0000)

/* Set up a "bad" return code

*/

CHGVAR &DC@RET 'N' MONMSG (CPF0000 MCH0000)

/* Return control to the calling program

*/

GOTO ENDPGM
/*=====================================	=======================================	=======================================
ENDPGM		

19.1.5 Example - Define a Built-In Function as a 3GL Subroutine

This section goes through the steps involved in creating a Built-In Function that will become part of the inline code of a LANSA function. The example is very simple but it will give you an understanding of how a Built-In Function fits into a LANSA system.

SCENARIO: This is a User Defined Built-In Function. The Built-In Function is used to flip dates from YYMMDD format to DDMMYY format.

The Built-in function will need some data structures (I SPECS) and some code (C SPECS) to drive the date manipulation.

The Built-In Function will be called UD_FLIP_DATE.

The next available identifier is 412.

BIF Name:	UD_FLIP_DATE
Unique Identifier:	412
Description:	Change date from YYMMDD to DDMMYY
Call or Execute:	Ε
Program Name:	IB@412
Terminates between calls:	Ν
Number of arguments:	1
Number of return values:	1

BIF Definition (as per DC@F47 file)

Argument List (as per DC@F48 file)

<u> </u>	-
BIF Name:	UD_FLIP_DATE
Parameter Type	ARG
Parameter Sequence:	1
Parameter No:	1
Parameter Ident:	А

Description:	Date in YYMMDD
Required / Optional:	R (R or O)
Parameter Type:	N (A, N or L)
Minimum Length:	6
Maximum Length:	6
Minimum Decimal:	
Maximum Decimal:	
Pass Length:	6
Pass Decimal:	
Default:	

Return Values (as per DC@F48 file)

BIF Name:	UD_FLIP_DATE
Parameter Type	RET
Parameter Sequence:	1
Parameter Number:	2
Parameter Identifier:	В
Description:	Date out (DDMMYY)
Required / Optional:	R (R or O)
Parameter Type:	N (A, N or L)
Minimum Length:	6
Maximum Length:	6
Minimum Decimal:	
Maximum Decimal:	
Pass Length:	6

rass Decilial.	Pass Decimal:	
----------------	---------------	--

Enter the data into the Built-In Function definition files: Create a member in QBIFSRC called BIF412ISPC and write the code for the data structures:

I* Code used	l by bui	lt-in fun	ction 412 - UD_FLIP_DATE	
I*	U			
IDATEIN	DS			
Ι		1	60NUMDI	
Ι		1	20YRIN	
Ι		3	40MNIN	
Ι		5	60DYIN	
I*				
IDATEOU	DS			
Ι		1	60NUMDO	
Ι		1	20DYOU	
Ι		3	40MNOU	
Ι		5	60YROU	

Create a member in QBIFSRC called BIF412CSPC and write the code to manipulate the dates.

C*============ _____ C* IB@412 : UD FLIP DATE CSR IB@412 BEGSR C^* MOVEL'Y' С DC@RET C^* С Z-ADDB@412A NUMDI C* С MOVELDYIN DYOU С **MOVELMNIN MNOU** С MOVELYRIN YROU C* С Z-ADDNUMDO B@412B C* CSR ENDSR

These source members will be included in the inline code of a LANSA function that uses this Built-In Function by /COPY commands generated at compile time.

19.1.6 Package 3GL Built-In Functions for Distribution

Create duplicate objects of any Built-In Function objects from your library to library QTEMP.

For example:

```
CRTDUPOBJ OBJ(OV@Pnnn)
FROMLIB(<bif pgms>)
OBJTYPE(*PGM)
TOLIB(QTEMP)
where <bif pgms> = the library where
the BIF objects
are stored
```

Copy relevant records from DC@F47 and DC@F48 to temporary versions of these files in QTEMP.

For example:

```
CPYF FROMFILE(<pgmlib>/DC@F47)
TOFILE(QTEMP/DC@F47)
MBROPT(*ADD)
CRTFILE(*YES)
INCREL((*IF F47BIF *EQ OV_XXXXXXXXXXX)
(*OR F47BIF *EQ OV_YYYYYYYYYY))
```

```
CPYF FROMFILE(<pgmlib>/DC@F48)
TOFILE(QTEMP/DC@F48)
MBROPT(*ADD)
CRTFILE(*YES)
INCREL((*IF F47BIF *EQ OV_XXXXXXXXXXXX)
(*OR F47BIF *EQ OV_YYYYYYYYYY))
```

where

<pgmlib> = LANSA Program Library

Copy any BIFnnnXSPC source members from QBIFSRC file to a source file in QTEMP.

For example:

CPYF FROMFILE(QBIFSRC) TOFILE(QTEMP/QBIFSRC) FROMMBR(BIFnnn*) TOMBR(*FROMMBR) MBROPT(*ADD) CRTFILE(*YES) FMT(*MAP *DROP) Save QTEMP to save media.

19.1.7 Install Other Vendor 3GL Built-In Functions

Restore objects from save media to QTEMP.

Create duplicate objects of any Built-In Function objects from library QTEMP to your Built-In Function library.

For example:

```
CRTDUPOBJ OBJ(OV@Pnnn)
FROMLIB(QTEMP)
OBJTYPE(*PGM)
TOLIB(<bif pgms>)
```

where

<BIF pgms> = the library where the BIF objects are stored.

```
Copy all records from DC@F47 and DC@F48 from QTEMP to the LANSA Program Library version of these files.
```

For example:

```
CPYF FROMFILE(QTEMP/DC@F47)
TOFILE(<pgmlib>/DC@F47)
MBROPT(*ADD)
CPYF FROMFILE(QTEMP/DC@F48)
TOFILE(<pgmlib>/DC@F48)
MBROPT(*ADD) FMT(*MAP *DROP)
<pgmlib> = LANSA Program Library
```

Copy all source members from source file in QTEMP to your Built-In Function source file (QBIFSRC).

For example:

```
CPYF FROMFILE(QTEMP/QBIFSRC)
TOFILE(QBIFSRC)
FROMMBR(*ALL)
TOMBR(*FROMMBR)
MBROPT(*ADD)
```

19.2 Create Your Own 3GL Built-In Functions on Windows/Linux/IBM i

Note: On IBM i, these Built-In Functions are for RDMLX functions and components.

Before you start creating your 3GL BIF's, you must become familiar with the rules and conventions which are defined in the following topics:

- 19.2.1 Assumptions
- 19.2.2 Warnings
- 19.2.3 Naming Conventions for 3GL BIFs on Windows/Linux
- 19.2.4 Steps to Create 3GL Built-In Functions on Windows/Linux/IBM i
- 19.2.5 Example A Simple Averaging Function
- 19.2.6 Example Manipulate an RDML Working List
- 19.2.7 Example Manipulate an RDMLX Working List
- 19.2.8 Shipped U_BIF Macros
- 19.2.9 U_BIF Macros used in Static or Internal Functions

19.2.1 Assumptions

The following topics assume that you:

- will implement your Built-In Functions using C or C++.
- have some basic knowledge of C or C++.
- will compile and link your Built-In Function using the recommended compiler for the target platform(s) and linker as per previous recommendations in this guide.

If these assumptions are not correct, then it is assumed that you have an experienced resource available that will be able to understand the following topics and then "extrapolate" or "interface" them into whatever tool, language or compiler that you wish to use.

It is of particular importance that the naming conventions are adhered to, as failure to do so could result in unexpected application failure or portability problems.

Also see

19.2.2 Warnings

19.2.2 Warnings

Before using the following BIF examples, please read these warnings:

• The supplied examples are in C and have been tested using Microsoft Visual C/C++ 6.0. If you do not use one of these compilers, you may encounter complex errors that will waste a lot of your time.

Note: The version of MS compiler bundled with Visual LANSA is optimized for compiling objects within the LANSA development environment only. To compile your own C/C++ Built-In Functions, you may need to install a full Microsoft compiler, typically by installing an appropriate version of Microsoft's Visual Studio product.

- You, the developer, are TOTALLY and ABSOLUTELY responsible for the development, coding, testing, portability and maintainability of your user defined Built-In Functions. Although your LANSA distributor may be willing to help you design and develop user defined built-ins, they are not, in any way at all, obliged to do so.
- LANSA Pty Ltd, the developers of the LANSA products, are always available to help with the design, coding, testing and portability of user defined Built-In Functions. Quotes for such work may be obtained by contacting LANSA Support at lansasupport@LANSA.com.au.
- The Visual LANSA user defined Built-In Function facilities are much more powerful than the equivalent LANSA for iSeries facilities. When producing Built-In Functions that use advanced facilities under Visual LANSA (that must also run under IBM i in RPG/400) you should carefully ensure that, during the design phase, you can in fact actually implement such advanced features in RPG/400.

Please note this warning carefully as Visual LANSA implementation opens up a whole new set of possibilities because it is generated in C, using a more advanced architecture than the LANSA for iSeries RPG/400 implementation. When you create a Built-In Function that has to run under IBM i as well as under Windows and Linux, you must ensure that any Windows and Linux feature that you use can also be "emulated" under IBM i.

For example, in a Visual LANSA user defined Built-In Function you can receive "variable" working lists and then retrieve (at run time) the number of columns in the list and the names, types, lengths and decimal precisions of each column in the list.

No such capabilities exist in the RPG/400 version. RPG/400 largely

precludes the use of such "dynamic" facilities.

19.2.3 Naming Conventions for 3GL BIFs on Windows/Linux

Built-In Function Definition (BIF Name)

- BIF names can be up to 18 characters in length.
- The BIF name is composed of letters of the English alphabet (A through to Z), 0 through to 9 and underscore only.
- The BIF name should be in the format XX_YYYYYYYYYYYYY where

XX must be either UD for User-Defined or OV for Other Vendor

YYYYYYYYYYYYYY is the name that will describe even to the non-programmer what functions the BIF performs

Program name

• Program names can be up to 8 characters in length

The program (dll) name must be in the format U_BIFxxx.DLL

where

xxx is a unique identifier

999 is the unique identifier in the range:

400 to 600 for User-Defined (UD) BIFs.

600 to 999 for Other Vendor (OV) BIFs as per LANSA allocated

range.

19.2.4 Steps to Create 3GL Built-In Functions on Windows/Linux/IBM i

- 1. Design the Built-In Function by giving the Built-In Function program a name according to the 19.2.3 Naming Conventions for 3GL BIFs on Windows/Linux.
- Complete a Built-In Function definition form. You can print the example supplied in 19.4 Built-In Function Sample Form or one that you have designed yourself. Refer to an example of a 3GL BIF definition, such as 19.2.5 Example A Simple Averaging Function, to understand the composition of the BIF definition.

Note: This step is not compulsory but it is recommended, especially if you are creating your first Built-In Function. It is meant as an aid to translating your Built-In Function design into the necessary data.

3. Enter the relevant information into the Built-In Function definition files.

Note: The next two steps must be performed. If the Built-In Function is not described in these files then it will not be recognized by LANSA. If this data is incorrect, then the Built-In Function will not work as expected.

4. Create Definitions in LX_F47 and LX_F48

There are two available methods for creating Definitions in LX_F47 and LX_F48.

Method 1: Define the BIF on IBM i and port to Windows

- a. Define your Built-In Function into the IBM i based table DC@F47 as described in the 19.1 Create your own 3GL Built-In Functions on IBM i for RDML Functions.
- b. Define your Built-In Function's arguments and return values into the IBM i based table DC@F48 as described in the Examples in 19.1 Create your own 3GL Built-In Functions on IBM i for RDML Functions.
- c. Optionally, design, code and test an IBM i based version of your Built-In Function. The LANSA Built-In Function facility is designed to be a generic interface between LANSA applications and multiple platforms.

Note: If you need to supply Built-In Functions that will eventually be required on multiple platforms, you may consider producing fatal error "stubs" for your Built-In Function(s) on all supported platforms. This

means that if your Built-In Function is accidentally invoked on a nonsupported platform (e.g. the IBM i) then a neat "fatal error" explaining the situation will result, rather than a more complex problem involving compile and/or run time linkage problems.

- d. Export the definition of your Built-In Function from the IBM i Repository using the standard LANSA REQUEST(PCMAINT) facility.
- e. Import the definition of your Built-In Function into each Visual LANSA system that will need to use it. To do this, use the standard Visual LANSA import facilities from CD or shared folder. This facility will update SQL tables LX_F47 and LX_F48 with details of the Built-In Functions exported from the IBM i in step 4.

Method 2: Define the BIF via SQL directly in Windows

This is an alternative method, which can be used where the IBM i is nonexistent or if you are familiar with inserting and updating your database on the Windows/Linux platform using SQL statements. This method requires you to directly enter your definitions into the LX_F47 and LX_F48 files.

Note: Inserting/updating records in the LX_F47 and LX_F48 internal LANSA tables via SQL, if not completed correctly, can potentially cause database corruption. If you require assistance with inserting/updating records in the LX_F47 and LX_F48 internal LANSA files via SQL, you should consult your Database Administrator.

Example of an insert to file LX_F47

INSERT INTO "LX_DTA"."LX_F47" VALUES('UD_AVERAGE',413,'Ge Example of an insert to file LX_F48

INSERT INTO "LX_DTA"."LX_F48" VALUES('UD_AVERAGE','ARG',1

- 5. If you are writing a BIF specifically for Windows or you require 64 bit support on Windows then you may find the LANSA User Defined BIF Wizard an easier method than the remainder of these steps 6 9. Instead go to Define Built In Functions with a Visual Studio Wizard It is particularly useful for existing Visual Studio users and when requiring more than one user defined BIF.
- 6. Create copies of the following files that are installed in your

x_win95\x_lansa\source directory. Change the "NNN" portion of the name to the unique identifier assigned to your Built-In Function.

For example, if you were making Built-In Function 445, and working from the C: drive then you might use these commands:

CD \X_WIN95\X_LANSA\SOURCE COPY U_BIFNNN.* U_BIF445.*

These are the files that are installed in your x_win95xlansasource directory:

U	BIFNNN.C	A sample/skeleton of a user E	Built-In C program.
_		1	1 0

U_BIFNNN.MAK A sample/skeleton of a make file required to compile and link your C program with Microsoft Visual C/C++ into a DLL under the Windows Operating System.

- U_BIFNNN.DEF A sample/skeleton of a module definition file required to link your C program with Microsoft Visual C/C++ into a DLL under the Windows Operating System.
- u_bifnnn.unx A sample/skeleton of a make file required to compile and link your C program into a shared library under a supported Linux operating system.
- U_BIFNNN.ISB A sample/skeleton of a module definition file required to link your C program into a service program on IBM i.
- 7. Edit each of the copied files using a standard source editor such as E or EPM:

File Description Of Changes

.C Scan/Replace all "nnn" with "xxx" where xxx is the unique id. Include code appropriate for your requirement.

.MAK Only required if compiling with Microsoft Visual C/C++.

Replace "nnn" with "xxx" in the BIFNAME line, where xxx is the unique id.

.DEF Only required if compiling with Microsoft Visual C/C++. Replace "nnn" with "xxx" where xxx is the unique id. Replace xxxxxxx in DESCRIPTION with a short description of the **Built-In Function**

- .unx (Only if planning to execute this BIF on Linux.) Scan/Replace all "nnn" with "xxx" where xxx is the unique id.
- .ISB Only required if planning to execute this BIF on IBM i.) Replace "nnn" with "xxx" where xxx is the unique id. Replace xxxxxxx in the comment area (/* */) with a short description of the Built-In Function
- 8. Back up your .C, .MAK, .WMK, .DEF, .unx and .ISB files as soon as possible. This is source code and it needs to be backed up frequently. Accidental loss of source code is a very common problem on unsecured PC systems.
- 9. Compile your Built-In Function using one of the following examples depending on the platform you are using:

Compile your Built-In Function on Windows Compile your Built-In Function on Linux Compile your Built-In Function on IBM i

Compile your Built-In Function on Windows

Compile and link it into a DLL (you may need to widen this window to see the example properly):

When using the Microsoft C compiler shipped with LANSA, the PATH, INCLUDE and LIB environment variables need to be initialized with the location of this compiler. First, run the cmd program to open up a command prompt. Then set the environment variables using the following examples. You will need to change the directory name to reflect where you have installed LANSA:

set Path=C:\Program Files\LANSA\MicrosoftCompiler90\bin;%Path% set INCLUDE=C:\Program

Files\LANSA\MicrosoftCompiler90\include;%include% set LIB=C:\Program Files\LANSA\MicrosoftCompiler90\lib;%lib%

When using a Microsoft compiler direct from Microsoft, open a command window using the Microsoft-installed short cut similar to *Visual Studio Command Prompt (2010)* in the Microsoft Visual Studio 2010\Visual Studio Tools program folder. This sets up the required environment variables.

Change the current drive to be\X_LANSA\SOURCE

---> Edit the .C file via a standard source editor

To attempt a compile and link, for Microsoft C/C++ use Command: NMAKE /e /a /f U_BIFnnn.MAK SRCROOT=D:\LANSAV10\X_WIN95\X_LANSA TRGROOT=D:\LANSAV10\X_WIN95\X_LANSA See Command Notes below

---- If errors detected

If no errors, test by executing an RDML function that uses your built-in function.

Backup your .C, .MAK, .WMK, .DEF and .DLL files. (the DLL is created in the x_lansa/execute directory).

Command Notes:

In this command:

NMAKE /e /a /f U_BIFnnn.MAK SRCROOT=D:\LANSAV10\X_WIN95\X_LANSA TRGROOT=D:\LANSAV10\X_WIN95\X_LANSA

SRCROOT is the location of the SOURCE directory where the .C is located

TRGROOT is the location of the EXECUTE & OBJECT directories where the .OBJ & .EXE will be created.

The paths cannot contain directory names that contain spaces. This is a restriction of Microsoft's NMAKE utility. To obtain an alternative name (otherwise known as a short name) for the directory with a space in it, open a "Command Prompt" window and use the "/X" option with the DIR command (i.e. DIR /X). The short name will be displayed next to the long name. For example, the common windows directory "PROGRAM FILES" will typically have a short name of "PROGRA~1". Your NMAKE command may look like this:

NMAKE /e /a /f U_BIFnnn.MAK SRCROOT=D:\PROGRA~1\LANSA\X_WIN95\X_LANSA TRGROOT=D:\PROGRA~1\LANSA\X_WIN95\X_LANSA

Define Built In Functions with a Visual Studio Wizard

LANSA provides a Visual Studio Wizard to generate the project files and setup the compile options for your user defined BIF so that all you need to do is write the C code.

The primary environment supported for the Wizard is Visual Studio 2010 and Visual LANSA 13.2 and later. LANSA itself is built with Visual Studio 2010. So it is best to use the same compiler so that the same Visual Studio runtime is used.

The installation instructions are documented in:

Install the Wizard

The steps to create a project are documented in: Create a Visual Studio Project based on the Wizard

This template may be used with later Visual Studio versions. There are changes required which are documented in:

Support for Visual Studio 2012 and later.

This template should be usable with Visual LANSA 12.0 and later, but it has not been tested. There is one known change that is required to the lansax property sheet as documented in Modify DLL Version Information. If the generated project compiles without error, it is supported. Otherwise it is not. See also Support in Visual LANSA 13.0 and earlier.

Install the Wizard

There are three parts to the Visual Studio User Defined BIF Wizard, delivered in three directories in the LANSA Tools directory. They are:

- 1. LANSAUBIFWizard the main Wizard code.
- 2. LANSAUBIFProject declaration of the Wizard so that it appears in the Visual Studio New Project dialog.
- 3. LANSAUBIFSharedFiles property sheets to declare common properties between User Defined BIF projects and other files to declare common version information.

There are various options for how you install a Visual Studio Wizard, however, it is recommended that you deploy it in the same location as the other VisualC++ templates. This requires administrative access. There are two folders:

- {VS2010InstallationFolder}\VC\VCWizards: Hosts the wizard projects; you should deploy the wizard project in the AppWiz subfolder, simply copy the whole project here from {Visual LANSA Installation Folder}\Tools\LANSAUBIFWizard, including the directory itself so that you end up with this directory {VS2010InstallationFolder}\VC\VCWizards\AppWiz\LANSAUBIFWizard;
- {VS2010InstallationFolder}\VC\VCProjects: copy the whole directory CONTENTS here from {Visual LANSA Installation Folder}\Tools\LANSAUBIFProject

The Wizard is now available to create a Visual Studio project, but there are some ancillary steps to follow as described in Create a Visual Studio Project based on the Wizard.

Create a Visual Studio Project based on the Wizard

You will need to use an empty solution to contain the user defined BIF that the Wizard creates.

1. Open Visual Studio, click *New Project* and then navigate to the *Visual Studio Solutions Template* as shown below:



You must specify the location of the Solution as **x_win95****x_lansa****source** as shown above.

- 2. Click *OK*. The solution will be created.
- 3. Right click the solution entry and select "open folder in windows explorer"
- Copy all the files from {Visual LANSA Installation Folder}\Tools\LANSAUBIFSharedFiles into this folder.
- 5. Now create the UBIF project using the LANSA Wizard:



Ensure that the project name is U_BIFnnn where nnn is the BIF number. This name must be entered in table LX_F47 column X47PGM. For details, refer to 19.2.4 Steps to Create 3GL Built-In Functions on Windows/Linux/IBM i.

By following these instructions and placing the solution in the correct directory, header files and libraries will be located correctly and the resulting DLL will be in either the x_win95xlansa/execute directory or in the

x_win64\x_lansa\execute directory, ready to be debugged and executed with LANSA.

The template that creates the core C program logic contains an example from the documentation with some message boxes added so that it is easier to see it working - see Example - A Simple Averaging Function. To run it from RDML also requires data to be added to tables LX_F47 and LX_F48. Refer to Steps to Create 3GL Built-In Functions on Windows/Linux/IBM i for details on how to do this. Obviously the C code would need to be modified to achieve your desired result.

If you build the 64-bit version and it fails, probably with the following message: Link : fatal error LNK1104: cannot open file 'x_funms.lib'

then you do not have 64 bit support installed. This is available in Visual LANSA 13.2 and later and is an installation option.

Modify DLL Version Information

A Windows DLL may optionally contain descriptive information that may be viewed in Windows Explorer. Right-click the DLL, select *Properties* and then the *Details* tab.

The file that contains this information is lansaver.h. The easiest way to edit this file is by adding it to your Project. You should change all the version numbers to your application's version numbers and replace LANSA with your own company's name.

Note that the Product Name is declared in the lansax property sheet in the *Resources Preprocessor Definitions*. It is supplied with **VER_PROD_NAME = Visual LANSA**.

The critical attribute for distributing a change in a Windows Installer patch is LANSA_FILE_VER. This number needs to be incremented in order for the DLL to be included in a patch and then installed on the deployed system.

Support in Visual LANSA 13.0 and earlier

Support for the Wizard in Visual LANSA 13.0 and earlier requires a small modification to the lansax property sheet as shown in the following screen capture. Add the highlighted definitions. Make sure that you SAVE the project sheet so that the changes take effect.

Common Descention	Deserve and a Definitions		
General Control	Hereprocessor Demnitions	windsz;_windows;_oskDLL;\$(X_OS_DE	13);A_LA;A_
User Macros	Undefine Preprocessor Def	muons	
VC++ Directories	Undefine All Preprocessor	Preprocessor Definitions ?	×
4 C/C++	Ignore Standard Include P		
General	Preprocess to a File	X_OPERATING_SYSTEM_WIN_	^
Ontimization	Preprocess Suppress Line	X_OPERATING_SYSTEM_WIN95 WIN32	
Preprocessor	Keep Comments	WINDOWS	
Code Generation		USRDLL	
Language			×
Precompiled Heade			>
Output Files		Inherited values:	
Browse Information		WINDU	
Advanced			
Command Line			
Linker			
Manifest Tool			U .
> Resources			
MIDL		✓ Inherit from parent or project defaults Macr	os>>
> XML Document Genera			
Browse Information		OK Car	ncel
Build Events			
Custom Build Step			
Managed Resources	Preprocessor Definitions		
Custom Build Tool	Treprocessor Deminions		

Support for Visual Studio 2012 and later

LANSA recommends that you use the version of Visual Studio that the LANSA runtime is built with. For Visual LANSA 13.x that is Visual Studio 2010.

The LANSAUBIFWizard.vsz file needs to be modified to change the version to match the Wizard running it.

The shipped Wizard is for Visual Studio 2010 and is known internally as 10.0. Hence this line in the file:

Wizard=VsWizard.VsWizardEngine.10.0

Visual Studio 2012, is known internally as 11.0:

Wizard=VsWizard.VsWizardEngine.11.0

Visual Studio 2013 is known internally as 12.0:

Wizard=VsWizard.VsWizardEngine.12.0

The generated project requires Visual Studio 2010 to be installed. If it is not there will be an error when building it:

Error MSB8020

Do what the message says to fix it.

Compile your Built-In Function on Linux

Compile and link it into a shared library:

--> Edit the .C file via a standard source editor copy the .C and .UNX files to \$LANSAXROOT/x_lansa/source (Use lowercase filenames, e.g. u_bif445.c) cd \$LANSAXROOT/x_lansa/source For Linux : Use command -> make -f u_bifnnn.unx

If no errors, test by executing an RDML function that uses your built-in function.

Backup your .c, .unx and libu_bif* files. (libu_bif is created in the x_lansa/bin directory).

Compile your Built-In Function on IBM i

To compile and link it into a service program:



If no errors, test by executing an RDML function that uses your built-in function.

Backup your .c and .ISB files and your U_BIFnnn service program. (U_BIFnnn service program is created in the LANSA program library).

Tips & Techniques

- The resulting object is a DLL. It is called dynamically by your Visual LANSA application, so there is no need to recompile your Visual LANSA RDML function when you change your Built-In Function, but you should exit and restart your x_run "session", so that the new dynamic address can be resolved for the DLL.
- You should not recompile a DLL while it is active or in use.

The locking of an "in use" DLL is performed by the operating system. If you are testing a DLL and it fails many times it is possible that the operating system will get confused and indicate that the DLL is in use, even when it is not. To solve this problem, reboot your PC.

Typically this problem is indicated by some sort of "Cannot create run file" type of error during the compile and link process. The file is locked by the operating system and the compiler/linker cannot overwrite it.

• If your Built-In Function needs to work on Windows and Linux, try to use standard C functions. If you need to write code which is different (for example, to include different header files), use the following macros to distinguish between Windows and Linux code:

```
/* Windows code */
#ifdef X_OPERATING_SYSTEM_WIN
...
#endif
/* Standard code */
...
/* Linux code */
#ifdef X_OPERATING_SYSTEM_Linux
...
#endif
```

3GL Program Rules and Guidelines

The following rules and guidelines should be observed when coding User Defined Built-In Functions:

- Do not prefix any variable or #define value with X_, V_ or U_.
- Avoid using variable or #define names that conflict with any of the names and types defined in X_GLODEF.H , X_GLOUSR.H or X_BIF000.H.
- All U_BIFnnn.DLL's must reside in the\x_lansa\execute directory at the time they are to be executed.
- Use the Visual LANSA type definition variables X_SHORT, X_LONG, X_DOUBLE, X_CHAR and X_VCHAR to define working storage in your function whenever possible. You may also use X_PSHORT, X_PLONG, X_PDOUBLE, X_PCHAR and X_PVCHAR as pointer declarations to the same types.
- Use the VISUAL LANSA variable prefixes whenever possible. For example:

s is the prefix for X_SHORT

l is the prefix for X_LONG

d is the prefix for X_DOUBLE

c is the prefix for X_CHAR

vch is the prefix for X_VCHAR

p is the prefix for a pointer

a is the prefix for an array

Prefixes can accumulate, so "apsOperation" indicates that nominal variable "Operation" is an array of pointers to X_SHORTs.

Similarly pvchName indicates that this is a pointer to an X_VCHAR variable that contains a "Name".

- Use the U_BIF macros whenever possible to retrieve and return Built-In Function argument variables. This will go a long way towards insulating your application from future changes to the VISUAL LANSA code generation techniques.
- Use one Built-In Function per C source file and one per DLL.

This rule will making porting of your code to other platforms much easier. Multi-entry point DLLs (or shared objects) are much harder to address under some Linux systems, so avoid hitting upon this problem later when you come to port your Built-In Functions.

- Most user defined Built-In Functions reference various .H header files shipped with Visual LANSA. Do not change these files as they are replaced every time a new version of VISUAL LANSA is installed and all your changes will be lost.
- A user defined Built-In Function has the following arguments:

struct X_IDS *pX_Ids,

struct X_PRO *pX_Pro,

struct X_FUN *pX_Fun,

struct X_BIF *pX_Bif,

```
struct X_FLD X_Fld[],
```

struct X_LIST X_List[],

X_SHORT sInCount,

X_SHORT sInVec[],

X_SHORT sRetCount,

X_SHORT sRetVec[]

Modification of storage pointed to by any of these arguments, other than by U_BIF macros, may result in unpredictable results and application failure. Do not modify storage outside the direct scope of your function other than by using the U_BIF macros. Failure to observe this rule may lead to loss of integrity in your application and future compatibility problems.

19.2.5 Example - A Simple Averaging Function

Defining a Built-In Function as a 3GL program

This section goes through the steps involved in creating a Built-In Function as a program. The example shown is very simple, but should give an insight to how LANSA uses a Built-In Function.

SCENARIO: This is a User Defined Built-In Function. The Built-In Function is used to retrieve system values.

A simple C program. The program will be passed the system value name and will return the system value.

The name will be UD_AVERAGE

The next available identifier at this site is 413.

This Built-In Function might be accessed in the LANSA RDML language like this:

```
define field(#number1) type(*dec) length(7)
define field(#number2) type(*dec) length(7)
define field(#mean) type(*dec) length(7)
```

request fields(#number1 #number2)

```
use ud_average with_arg(#number1 #number2) to_get(#mean)
pop_up_fields(#mean)
```

Following is a C user-defined Built-In Function definition called UD_AVERAGE that receives 2 decimal values as arguments and returns their average value.

All arguments and return values are mandatory values.

While this Built-In Function is trivial and the example of its use is trivial, it is a simple starting point:

BIF Definition (as per DC@F47 file on IBM i/ LX_F47 file on Windows)

BIF Name:	UD_AVERAGE
Unique Identifier:	413
Description:	Retrieve a system value
Call or Execute:	С
Program Name:	U_BIF413

Terminates between calls:	Ν	(Y or N)
Number of arguments:	2	
Number of return values:	1	

BIF Arguments (as per DC@F47 file on IBM i/ LX_F47 file on Windows) Parameter 1

BIF Name:	UD_AVERAGE
Parameter Type	ARG
Parameter Sequence:	1
Parameter Number:	1
Parameter Identifier:	А
Description:	First Value
Required / Optional:	R (R or O)
Parameter Type:	N (A, N or L)
Minimum Length:	7
Maximum Length:	7
Minimum Decimal:	
Maximum Decimal:	
Pass Length:	10
Pass Decimal:	
Default:	

Parameter 2

BIF Name:	UD_AVERAGE

Parameter Type	ARG
Parameter Sequence:	2
Parameter Number:	2
Parameter Identifier:	В
Description:	Second Value
Required / Optional:	R (R or O)
Parameter Type:	N (A, N or L)
Minimum Length:	7
Maximum Length:	7
Minimum Decimal:	
Maximum Decimal:	
Pass Length:	10
Pass Decimal:	
Default:	

BIF Return Values (as per DC@F47 file on IBM i/ LX_F47 file on Windows)

BIF Name:	UD_AVERAGE	
Parameter Type	RET	
Parameter Sequence:	1	
Parameter Number:	3	
Parameter Identifier:	С	
Description:	Mean Value	
Required / Optional:	R (R or O)	
Parameter Type:	A (A, N or L)	

Minimum Length:	7
Maximum Length:	7
Minimum Decimal:	
Maximum Decimal:	
Pass Length:	256
Pass Decimal:	

Now, enter the data into the Built-In Function definition files:

/* _____ /* ======= USER DEFINED BUILT-IN FUNCTION DEFINITION ======= */ */ /* /* This is a sample of how a user defined built-in 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 built-in functions are a powerful facility. However,*/ /* you should note that YOU are responsible for any impact the */ /* use of a user defined built-in function has on the performance, */ /* security, integrity, portability and maintainability of your */ /* applications. */ /* /* ______ /* */ : U_BIF413.C /* Source File */ /* Entry Point Name : U Builtin 413 /* Linked DLL Name : U BIF413.DLL */ /* Shared Object Name : u bif413.0 /* OS/Dependencies */ : Yes/No /* */

```
/* Amendment History :
                                            */
/*
                                    */
                                            */
/* Task Id Date Description
/* ====== ====
                                                   */
                  ============
/*
                                    */
/* ______
#define U BIF FUNCTION
                           U Builtin 413
#define U_BIF_FUNCTION_NAME "U_Builtin_413"
#define U_BIF_DESCRIPTION "This is a description of this built-in"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "x_glodef.h"
#include "x_glousr.h"
#ifdef X OPERATING SYSTEM WIN
#include <windows.h>
#endif
#include "x funstr.h"
#include "x_funpro.h"
#include "x_bif000.h"
/*===
/*
                                    */
/* Arguments : pX_Ids
                        - Standard X IDS system definition
                                                        */
                    - Standard X PRO process definition */
/*
          pX Pro
          pX_Fun
/*
                    - Standard X FUN function definition */
/*
          pX Bif
                   - Standard X BIF built-in definition */
/*
          X Fld[:
                   - Standard X FLD field definitions
                                                   */
/*
          X List[:
                   - Standard X LIST list definitions
                                                  */
                                                   */
/*
          sInCount - Number of arguments passed in
                   - Vectors of arguments
/*
          sInVec[:
                                              */
/*
          sRetCount - Number of return values
                                                 */
/*
          sRetVec[: - Vectors of return values
                                               */
/*______
                                               _____
```
X_VOID_FUNCTION U_BIF_FUNCTION (U_BIF_STANDARD_PARAM {

/* ______ */ /* Handle a shutdown request (usually no activity is required) */ /* _____ */ if (U_BIF_SHUTDOWN_REQUEST) { U_BIF_SET_GOOD_RETURN } /* _____ */ /* Else perform the requested activity */ /* ______ */ else { X_LONG lArg1; X LONG lArg2; X_LONG lAverage; /* _____ */ /* Get argument 1 (C convention is 0) */ /* and argument 2 (C convention is 1) */ /* and compute their average */ /* _____ */ U BIF GET ARG AS LONG (0, lArg1) U_BIF_GET_ARG_AS_LONG (1,lArg2) lAverage = (lArg1 + lArg2) / 2;/* ______*/ /* Return the result in return value 1 */ /* (C convention is 0) */ /* _____* */ U_BIF_SET_RET_FROM_LONG (0, lAverage); /* _____ */ /* Set a "good" return (Operating Level) */

/* ______ */



Some significant things to note about this example are:

• The very extensive use of C macros. All the U_BIF_xxxx references are to macros contained in header (.H) file X_BIF000.H.

These are expanded at compile time into C code. By using these wherever possible as your "interface" between your Built-In Function and the Visual LANSA definitions and call interface, you can protect your application from future changes.

There are a number of U_BIF macros available. Refer to 19.2.8 Shipped U_BIF Macros for a complete list.

• The definition of local variables by using types such as X_LONG. These definitions are contained in the standard header file X_GLODEF.H and are shipped with Visual LANSA. Variables passed into U_BIF macros should always be declared using these types (as indicated in the macro definition):

Variable Declaration	Pointer Declaration	Actual C Type involved
X_VOID	X_PVOID	Void
X_BOOL	X_PBOOL	Int
X_SHORT	X_PSHORT	Short
X_USHORT	X_PUSHORT	unsigned short
X_LONG	X_PLONG	Long
X_ULONG	X_PULONG	unsigned long
X_LONGLONG	X_PLONGLONG	LONGLONG in Windows: 64 bits signed Integer. Not recommended if used with WATCOMC

X_DOUBLE	X_PDOUBLE	double
X_CHAR	X_PCHAR	char (single character only)
X_VCHAR	X_PVCHAR	char (> 1 character with a null/end of string terminator)

X_LIST_COUNT X_LIST_COUNT Long

• The use of the C numbering convention. By convention most C objects are numbered by offset (i.e. "N" things are numbered from 0 to N-1).

This convention is used in all the U_BIF macros. So "Argument Number 3" is actually referenced in U_BIF macros as number 2.

Generally this approach makes coding easier. For example, you might make this function more generic and receive from 2 to 20 numbers as arguments (i.e. no zero divide test is required). To do this you might change the body of the function to be like this:

```
{
X_LONG lArgValue;
X_SHORT sTotalArgs;
X_SHORT sCurrentArg;
X_DOUBLE dTotal = 0;
```

```
U_BIF_GET_ARGUMENT_COUNT (sTotalArgs)
```

```
for (sCurrentArg = 0; sCurrentArg < sTotalArgs; sCurrentArg++)
{
    U_BIF_GET_ARG_AS_LONG (sCurrentArg, lArgValue)
    dTotal = dTotal + lArgValue;
}
lAverage = dTotal / sTotalArgs;
U_BIF_SET_RET_FROM_LONG (0, lAverage)</pre>
```

```
U_BIF_SET_GOOD_RETURN }
```

19.2.6 Example - Manipulate an RDML Working List

This second example is a simple variation on the first example.

It is called UD_AVERAGE_LIST and it allows an RDML "working list" of numbers to be passed into it.

All the values in the list are accumulated to produce the average value.

This function's unique identifier is 414.

It has one mandatory argument a working list of numbers.

It has one mandatory return value the average of the list of numbers.

Used in an RDML function like this it could support the averaging of up to 1000 entries:

```
define field(#number) type(*dec) length(7)
def_list name(#list) fields(#number) type(*working)
        entrys(1000)
define field(#mean) type(*dec) length(7)
```

```
use ud_average_list with_arg(#list) to_get(#mean)
```

An example of the user defined Built-In Function required to implement UD_AVERAGE_LIST might be coded like this:

```
/ ______
/* ======= USER DEFINED BUILT-
IN FUNCTION DEFINITION ======= */
/* _______
/*
                       */
/* Source File
                             */
            : U BIF414.C
/* Entry Point Name
                               */
              : U Builtin 414
/* Linked DLL Name
               : U BIF414.DLL
                                 */
/* Shared Object Name
               : u bif414.0
                               */
/* OS/Dependencies
              : Yes/No
                              */
                       */
/*
/* Amendment History :
                            */
/*
                       */
/* Task Id Date Description
                            */
*/
/*
                       */
```

```
#define U_BIF_FUNCTION U_Builtin_414
#define U_BIF_FUNCTION_NAME "U_Builtin_414"
#define U_BIF_DESCRIPTION "This is a description of this built-in"
```

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

#include "x_glodef.h"
#include "x_glousr.h"

#ifdef X_OPERATING_SYSTEM_WIN
#include <windows.h>
#endif

#include "x_funstr.h"
#include "x_funpro.h"
#include "x_bif000.h"

#incl /*	ude "x_bif000.h"
/* /*	*/
/* Ai	rguments : pX_Ids - Standard X_IDS system definition */
/*	pX_Pro - Standard X_PRO process definition */
/*	pX_Fun - Standard X_FUN function definition */
/*	pX_Bif - Standard X_BIF built-in definition */
/*	X_Fld[: - Standard X_FLD field definitions */
/*	X_List[: - Standard X_LIST list definitions */
/*	sInCount - Number of arguments passed */
/*	sInVec[: - Vectors of arguments */
/*	sRetCount - Number of return values */
/*	sRetVec[: - Vectors of return values */
/*	*/
/*==	=======================================
Χ٧	OID FUNCTION U BIF FUNCTION (U BIF STANDARD PARAM
{ _	
C C	
/* .	*/
/*]	Uandle a chutdown request (usually no activity is required) */

/* Handle a shutdown request (usually no activity is required) */

/* _____ */

```
if (U_BIF_SHUTDOWN_REQUEST)
{
 U_BIF_SET_GOOD_RETURN
}
/* ______ */
/* Else perform the requested activity
                                    */
/* _____ */
else
{
 U_BIF_DECLARE_LIST_POINTER (pListArg1)
 X_SHORT sEntrys = 0;
 X_LONG lAverage = 0;
 /* _____ */
 /* Set list pointer and get entry count */
 /* _____ */
 U_BIF_SET_ARG_LIST_POINTER (pListArg1, 0)
 U_BIF_GET_LIST_CURRENT_ENTRYS (pListArg1, sEntrys)
 /* _____ */
 /* If there is any entries in the list */
 /* _____ */
 if (sEntrys > 0)
 {
  X SHORT sCurrentEntry;
  X CHAR chFound;
  X LONG lValue;
  X DOUBLE dTotal = 0;
  /* ______*/
  /* Process all list entries and calculate */
  /* the average value of all of them
                            */
  /* _____ */
```

for (sCurrentEntry = 1; sCurrentEntry <= sEntrys; sCurrentEntry++)
{</pre>

```
U_BIF_GET_ENTRY_FROM_LIST (pListArg1, sCurrentEntry, chFou
    U_BIF_GET_LIST_COLUMN_AS_LONG (pListArg1, 0, lValue)
    dTotal = dTotal + lValue;
   }
   lAverage = dTotal / sEntrys;
  }
  /* _____ */
  /* Return the result in return value 1 */
  /* _____ */
  U_BIF_SET_RET_FROM_LONG (0, lAverage);
  U_BIF_SET_GOOD_RETURN
 }
/* _____ */
U_BIF_RETURN;
}
```

Some significant things to note about this example are:

- The manipulation of a working list. Working lists passed into Built-In Functions can be read, updated, added to or cleared by using a few simple U_BIF macros.
- Note that unlike arguments and return values, working list entries are numbered from 1 to N, rather than from 0 to (N-1).

19.2.7 Example - Manipulate an RDMLX Working List

The previous example can be changed to manipulate RDMLX lists.

The X_ALLOW_10_0_STRUCTURES define must be disabled by a set of #ifdef ... #endif at the beginning of the C code, just before #include "x_bif000.h" :

```
#ifdef X_ALLOW_10_0_STRUCTURES
#undef X_ALLOW_10_0_STRUCTURES
#endif
```

Also the list entries's counter and list entry's index type must to be changed from X_SHORT to X_LIST_COUNT.

Used in an RDMLX function like this it could support the averaging of up to 999999 entries:

```
use ud_average_list with_arg(#list) to_get(#mean)
```

An example of the user defined Built-In Function required to implement UD_AVERAGE_LIST might be coded like this:

```
_____
/* ======= USER DEFINED BUILT-
IN FUNCTION DEFINITION ======= */
/* ______
/*
                           */
/* Source File
              : U_BIF414.C
                                  */
/* Entry Point Name : U BuiltIn 414
                                    */
/* Linked DLL Name
                 : U BIF414.DLL
                                       */
/* Shared Object Name
                 : u bif414.0
                                    */
/* OS/Dependencies
                 : Yes/No
                                   */
/*
                           */
/* Amendment History :
                                 */
                           */
                                 */
/* Task Id Date Description
```

```
*/
/*
                                 */
/* ______
                         U BuiltIn 414
#define U BIF FUNCTION
#define U BIF FUNCTION_NAME "U_BuiltIn_414"
#define U_BIF_DESCRIPTION "This is a description of this built-in"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "x_glodef.h"
#include "x_glousr.h"
#ifdef X OPERATING SYSTEM WIN
#include <windows.h>
#endif
#include "x_funstr.h"
#include "x_funpro.h"
/* Enable RDMLX list*/
#ifdef X_ALLOW_10_0_STRUCTURES
#undef X_ALLOW_10_0_STRUCTURES
#endif
#include "x bif000.h"
/*
                                 */
/* Arguments
                      - Standard X IDS system definition
            :pX Ids
                                                   */
         pX_Pro
/*
                  - Standard X PRO process definition */
/*
         pX Fun - Standard X FUN function definition */
         pX_Bif
/*
                  - Standard X BIF built-in definition */
         X Fld[:
/*
                 - Standard X FLD field definitions */
         X_List[:
/*
                  - Standard X LIST list definitions */
/*
         sInCount - Number of arguments passed */
                - Vectors of arguments
/*
         sInVec[:
                                          */
/*
         sRetCount - Number of return values
                                             */
```



```
X_LONG lValue;
  X DOUBLE dTotal = 0;
  /* ______*/
  /* Process all list entries and calculate */
  /* the average value of all of them
                              */
  /* _____ */
 for (lCurrentEntry = 1; lCurrentEntry <= lEntrys; lCurrentEntry ++)</pre>
  {
   U_BIF_GET_ENTRY_FROM_LIST (pListArg1, lCurrentEntry, chFoui
   U_BIF_GET_LIST_COLUMN_AS_LONG (pListArg1, 0, lValue)
   dTotal = dTotal + lValue;
  }
   lAverage = dTotal / lEntrys;
  }
 /* _____ */
 /* Return the result in return value 1 */
 /* _____ */
  U_BIF_SET_RET_FROM_LONG (0, lAverage);
  U_BIF_SET_GOOD_RETURN
}
/* _____ */
/* Return control to caller
                                   */
/* ______ */
U BIF RETURN;
```

}

19.2.8 Shipped U_BIF Macros

To assist you in coding User Defined Built-In Functions, the following U_BIF macros are shipped with Visual LANSA.

Please note that these macros were specifically designed to make programming user defined Built-In Functions easier, and to insulate them from future changes.

They do not have a high degree of error handling and are not designed to be "bullet proof". It is assumed that the developer using them will be capable and will understand how to use them effectively.

The U_BIF macros are defined in header file X_BIF000.H that is supplied on the LANSA Windows product CD. Do not change this file as it is totally replaced every time that Visual LANSA is upgraded and your changes will be lost.

Note that the arguments must be specified in the order they are listed in the Arguments column.

Macro Name	Arguments	Descript
U_BIF_COLUMN_ERROR	None	Fatal errc
U_BIF_LIST_ERROR	None	Fatal errc
U_BIF_LANSA_FUNCTION	None	Name of
U_BIF_LANSA_FUNCTION_DESC	None	Descripti
U_BIF_LANSA_PROCESS	None	Name of
U_BIF_LANSA_PROCESS_DESC	None	Descripti
U-BIF-GET_HAB	_R=HAB	Returns c
U-BIF-GET_HMQ	_R=HMQ	Returns c
U_BIF_GET_HWND	_R = HWND	Returns c
U_BIF_FATAL_ERROR	None	Tests for (U_BIF_
U_BIF_ISSUE_FATAL_ERROR	_T = "text" or X_PVCHAR	Issues a f the caller
U_BIF_HANDLE_FATAL_ERROR	None	Tests for

		return if (
U_BIF_SET_GOOD_RETURN	None	Indicates calling R
U_BIF_SET_BAD_RETURN	None	Indicates calling R
U_BIF_RETURN	None	Returns c function.
U_BIF_SHUTDOWN_REQUEST	None	Tests for request a: (U_BIF_
U_BIF_STANDARD_PARAMETER	RS None	Defines t user defiı
U_BIF_STANDARD_ARGUMENT	S None	Defines t be passec Function
U_BIF_OPERATING_SYSTEM_WI	IN95 None	Use in a (compilati
U_BIF_OPERATING_SYSTEM_WI	IN None	Use in a (compilati 9x/200x.
U_BIF_OPERATING_SYSTEM_UN	NIX None	Use in a compilati operating
U_BIF_GET_ARGUMENT_COUN	T _N = X_SHORT	Returns a argument
U_BIF_ARGUMENT_PASSED	_N = X_SHORT	Used in a argument caller.
U_BIF_GET_RETURN_COUNT	$_N = X_SHORT$	Returns a return va
U_BIF_RETURN_REQUIRED	N = X SHORT	Used in a

return val caller. _N = X_SHORT Retrieves U_BIF_GET_ARG_AS_SHORT X_SHOF S = XSHORT or argum changed | argument from 0. U_BIF_GET_ARG_AS_LONG N = X_SHORT Retrieves X LONC L = X LONGor argum changed | argument from 0. U_BIF_GET_ARG_AS_DOUBLE N = X_SHORT Retrieves X_DOUI D = invalid, o X_DOUBLE _D is not the argun numbered U_BIF_GET_ARG_AS_VCHAR N = X_SHORT Retrieves $V = X_VCHAR X_VCHA$ or argum changed | argument from 0. U_BIF_SET_RET_FROM_SHORT N = X SHORTSets retur X_SHOF S = X SHORT invalid, o the entire return val numbered _N = X_SHORT Sets retur U BIF SET RET FROM LONG X LONC L = X LONGinvalid, o

			the entire return va numbere
U_BIF_SET_RET_FROM_	DOUBLE	_N = X_SHORT _D = X_DOUBLE	Sets return X_DOUI is invalid required, is the return numbered
U_BIF_SET_RET_FROM_	VCHAR	_N = X_SHORT _V = X_VCHAR	Sets return X_VCHA invalid, c the entire return va numberee
U_BIF_DECLARE_LIST_I	POINTER	_L = C Name	Declares received This is a be position pListArg recomment A declared any attent any way.
U_BIF_SET_ARG_LIST_P	OINTER	_L = C Name _A = X_SHORT	Initialized previousl U_BIF_I _A is the Function working If _A is in passed, a the Built
U_BIF_SET_RET_LIST_P	DINTER	_L = C Name _A = X_SHORT	Initialize previousl

			U_BIF_I _A is the In Functi working value _A result, ter
U_	_BIF_GET_ENTRY_FROM_LIST	_L = C Name _E = X_LIST_COUNT _R = X_CHAR	Retrieves working was foun otherwise entries ar different number i
U_	_BIF_CLEAR_LIST	_L = C Name	Clears all by _L.
U_	_BIF_ADD_ENTRY_TO_LIST	_L = C Name	Adds a noL.
U_	_BIF_UPDATE_ENTRY_IN_LIST	_L = C Name	Updates t list pointe
U_	_BIF_GET_LIST_CURRENT_ENTRYS	_L = C Name _R = X_LIST_COUNT	Returns i number o pointed to
U_	_BIF_GET_LIST_MAXIMUM_ENTRYS	_L = C Name _R = X_LIST_COUNT	Returns i entries al to by _L.
U_	_BIF_GET_LIST_ENTRY_LENGTH	_L = C Name _R = X_LIST_COUNT	Returns i of an entr by _L.
U_	_BIF_GET_LIST_COLUMN_TOTAL	_L = C Name _R = X_SHORT	Returns i columns pointed to
U_	_BIF_GET_LIST_COLUMN_ATTRIBS	_L = C Name	Queries t _C (in 0 ·

_C = X_SHORT pointed to T = X CHAR $_T$: The $_E = X_ULONG _E : The$ $_D = X_SHORT _D : The$ $_B = X_SHORT _B : The$ _T is retu defined in X_TYPE X_TYPE X_TYPE X_TYPE X_TYPE X_TYPE X_TYPE X_TYPE format, X_TYPE X_TYPE X_TYPE X_TYPE X_TYPE VarBinar X_TYPE X_TYPE If C is a error will Function. For X_T _E is the name can

U_BIF_LIST_COLUMN_NAME_LEN

None

Defines t

		currently
U_BIF_GET_LIST_COLUMN_NAME	_L = C Name _C = X_SHORT _R = X_VCHAR	Queries t _C (in 0 pointed to the colun _R must be at leas (U_BIF_ bytes lon If _C is a error will Function.
U_BIF_GET_LIST_COLUMN_AS_SHORT	_L = C Name _C = X_SHORT _R = X_SHORT	Returns t (in 0 -> (of the wo value is r type X_S column n terminati
U_BIF_GET_LIST_COLUMN_AS_LONG	_L = C Name _C = X_SHORT _R = X_LONG	Returns t (in 0 -> (of the wo value is r type X_L number a terminati
U_BIF_GET_LIST_COLUMN_AS_LONGLON	G _L = C Name _C = X_SHORT _R = X_LONGLONG	Returns t (in 0 -> (of the wo value is r type X_L column n terminati
U_BIF_GET_LIST_COLUMN_AS_DOUBLE		Returns t (in 0 -> (

		of the wo value is r type X_E column n terminati
U_BIF_GET_LIST_COLUMN_AS_VCHAR	_L = C Name _C = X_SHORT _R = X_VCHAR	Returns t (in 0 -> (of the wo The value of type X column n terminati
U_BIF_SET_LIST_COLUMN_FROM_SHORT	_L = C Name _C = X_SHORT _R = X_SHORT	Sets the v -> (N-1) working set from X_SHOF number a terminati current en inserted c applied to
U_BIF_SET_LIST_COLUMN_FROM_LONG	_L = C Name _C = X_SHORT _R = X_LONG	Sets the v -> (N-1) working set from X_LONC number a terminati current ei inserted c applied to
U_BIF_SET_LIST_COLUMN_FROM_LONG	_L = C Name _C = X_SHORT _R = X_LONGLONG	Sets the v -> (N-1) working set from

X_LONC column n terminati current ei inserted c applied to U_BIF_SET_LIST_COLUMN_FROM_DOUBLE _L = C Name Sets the v -> (N-1) : $_C = X_SHORT$ working] _R = set from X_DOUBLE X_DOUI number a terminati current ei inserted c applied to U_BIF_SET_LIST_COLUMN_FROM_VCHAR _L = C Name Sets the v -> (N-1) : $_C = X_SHORT$ working] $R = X_VCHAR$ set from X_VCH/ number a terminati current ei inserted c applied to

19.2.9 U BIF Macros used in Static or Internal Functions

The U_BIF macros are designed to only reference variables that are passed into them and from the standard Built-In Function parameter list U_BIF_STANDARD_PARAMETERS.

This can create an apparent problem if you try to use the U_BIF macros in a static or internal function within your C program.

For example you might define a static/internal function within your C program like this:

```
static void vAverage (X_LONG lArg1,
            X_LONG lArg2,
            1
{
   X LONG lAverage;
   lAverage = (lArg1 + lArg2) / 2;
   U BIF SET RET FROM LONG (sRetNo, lAverage);
   return;
}
```

To execute this function you might code:

```
vAverage (6, 42, 17);
```

If you do this you will find that it will not compile.

The U_BIF macro being used makes references to pX_Ids, pX_Pro, etc.

These are passed into the "mainline" routine of this function and are in scope there.

However, they are out of scope in this static/internal function and thus will cause compile errors.

You can correct this problem by changing the code to be like this:

```
static void vAverage (U_BIF_STANDARD_PARAMETERS,
            X LONG lArg1,
            X LONG lArg2,
            X SHORT sRetNo)
```

{

```
X LONG lAverage;
lAverage = (lArg1 + lArg2) / 2;
U BIF SET RET FROM LONG (sRetNo, lAverage);
```

return;

}

and then by changing the execution of the function to be like this:

```
vAverage (U_BIF_STANDARD_ARGUMENTS, 6, 42, 17);
```

Of course the behavior of some U_BIF macros is changed when they are used in a static or internal function.

For example, U_BIF_RETURN and U_BIF_ISSUE_FATAL_ERROR return control to the calling function in your program rather than to the calling RDML function.

This means that it is a good idea to test for a fatal error whenever you execute a static or internal function that uses U_BIF macros.

So in this example you could code:

```
vAverage (U_BIF_STANDARD_ARGUMENTS, 6, 42, 17);
U_BIF_HANDLE_FATAL_ERROR
```

19.3 Create Your Own RDML Built-In Functions (all platforms)

When creating RDML Built-In Functions for your own use or for distribution to other LANSA sites, there are some rules and conventions that you must follow. It is of particular importance that your RDML Built-In Functions comply with these rules and conventions as well as to the points in the 19.3.1 Conformance Checklist for Built-In Functions written in RDML

Please ensure that you check your Built-In Functions against the provided checklist to prevent unexpected application failure or portability problems.

See also

Built-In Functions in the Technical Reference Guide.

19.3.1 Conformance Checklist for Built-In Functions written in RDML

Please check that any Built-In Function that you write in RDML conforms to all the points in the following checklist.

Note: Failure to conform to all of these checklist points may, under some circumstances, lead to unexpected application failure, unexpected behavior and/or application portability problems.

- All Built-In Function argument and return value defaults are simple alpha or numeric literal values. Ensure no other type of default value (e.g. a system variable) has been used for arguments or return values.
- The Built-In Function name is simple. It is composed of letters of the English alphabet (A through to Z), 0 through to 9 and underscore only.
- The name of the RDML function containing the BIF logic is simple. It is composed of letters of the English alphabet (A through to Z), 0 through to 9, special characters @ and underscore only.
- The RDML function containing the BIF logic uses a specific *HEAVYUSAGE or *LIGHTUSAGE option in a FUNCTION command to indicate how it is to be (re)activated.
- The RDML function containing the BIF logic can handle a shutdown call (system variable *BIF_SHUTDOWN = Y) correctly regardless of whether it uses the *HEAVYUSAGE or *LIGHTUSAGE option. In most circumstances this simply involves this logic at the start of the function:

```
IF COND('*BIF_SHUTDOWN = Y')
RETURN
ENDIF
```

- No attempt is made in the RDML function containing the BIF logic to use the exchange list to either receive information from, or return information to, the function that has invoked the Built-In Function.
- The RDML function containing the Built-In Function logic terminates only with specific RETURN and ABORT commands. Other termination commands such as EXIT, MENU, TRANSFER, etc (regardless of whether they are used specifically or are generated by screen I/O commands) are not used to terminate the function.
- Any screen I/O commands used in the RDML function containing the BIF

logic (such as DISPLAY, REQUEST, POP_UP or MESSAGE) are conditioned by the job mode thus allowing the BIF to function viably in interactive or batch jobs

- The function routing table is not used in any way in conjunction with the RDML function containing the BIF logic. RDML BIF functions are not (re)routed by the function routing table. Refer to What is Function Routing in the *LANSA for iSeries User Guide*, if you'd like more details.
- Functions that invoke the BIF do not use the same working list in the WITH_ARG and TO_GET parameters of the USE command.
- No attempt is made to control access to the BIF using RDML function level security. BIFs coded in RDML are not subject to any form of invocation level security checking because of the severe performance overheads that this would incur.

19.3.2 RDML Naming Conventions

With RDML User-Defined Built-In Functions, you are not required to enter the definitions manually. The definitions are registered automatically when you compile your RDML Built-In Function.

Built-In Function Definition (BIF Name)

Unless overridden by the definition of field BIF_NAME within the function it defaults to the Function name. It is recommended that the following naming convention be adopted when overriding within the function.

- BIF names can be up to 18 characters in length.
- The BIF name is simple. It is composed of letters of the English alphabet (A through to Z), 0 through to 9 and underscore only.
- The BIF name should be in the format XX_YYYYYYYYYYYY where

XX must be either UD for User-Defined or OV for Other Vendor.

YYYYYYYYYYYYYY is the name that will describe, even to the non-programmer, what functions the BIF performs

Function name

• Function names can be up to 7 characters in length

While there aren't any mandatory naming conventions for RDML User-Defined Built-In functions, it is recommended that you follow these naming conventions, especially if you want it to be distributed:

• The name should be in the format XX@Y999

where

XX identifies this as Built-In Function code; i.e. OV for Other Vendor or UD for User-Defined.

y is the type of object, either P for Program or D for Display file

999 is the unique identifier in the range:

-400 to -600 for User-Defined (UD) BIFs. -600 to -999 for Other Vendor (OV) BIFs as per allocated range.

19.3.3 Built-In Function Argument

Inside the RDML Built In Function, an argument is signified by a field or working list defined with the name "BIF_ARGnn", where nn is between 01 and 20.

19.3.4 Built-In Function Return Value

Inside the RDML BIF, a return value is signified by a field or working list defined with the name "BIF_RETnn", where nn is between 01 and 20.

19.3.5 Steps to Create RDML Built-In Functions

- 1. Design the Built-In Function.
- 2. Allocate the next Unique number.

If this is a User Defined Built-In Function, this step is not compulsory, but if you do decide to take this step, use the next available number between -401 and -600.

If this is an Other Vendor Built-In Function then use the next available number in The LANSA Group allocated range (within the Other Vendor range of -601 to -999).

Once the number range is allocated by LANSA Pty Ltd then it would be necessary to reflect this into the DC@F47 & DC@F48 records for the OV RDML BIF(s) via DFU. Another possibility is to put in a dummy DC@F47 record that is one less than the start of your range, either before compiling the first OV RDML BIF the first time or removing the DC@F47/DC@F48 records for your already compiled OV RDML BIFs.

- 3. Give the Built-In Function a name. Use something that even a nonprogrammer would understand. Prefix your own (User Defined) BIFs with 'UD_' and Other Vendor BIFs with 'OV_' for easy identification as described in 19.3.2 RDML Naming Conventions.
- 4. Give the function a name. If the RDML BIF is to be distributed, this name must be as documented under naming conventions. If the BIF name is less than 7 characters and this is a User Defined BIF, then the function name could be the same as the BIF name. Refer to the 19.3.2 RDML Naming Conventions if necessary.
- 5. Complete a Built-In Function definition form. Use either the one that is described later in this manual, or one that you have designed yourself.

Note: This step is not compulsory but it is a good idea if this is your first Built-In Function. It is meant as an aid to translating your Built-In Function design into the RDML necessary to cause LANSA to define it for you. If this data is incorrect, then the Built-In Function may not work as expected.

6. Code your function. Use the BBRDMLBIF template supplied to assist with initial coding of the BIF definition, arguments and return values.

The application template called BBRDMLBIF will generate the function skeleton for an RDML Built-In Function. The RDML that is generated

follows the recommendations in 19.3.7 RDML Built-In Function Tips and Techniques.

19.3.6 How an RDML Built-In Function defines the BIF

A function is recognized as an RDML Built-In Function if it has function option *BUILTIN specified, along with *DIRECT. The BIF definitions are defined in the Definition File Layouts.

Definition File Layouts

The BIF definitions are defined in these Files: DC@F47 - Built-In Function Definition DC@F48 - Built-In Function Argument/Return Value Definition

DC@F47 - Built-In Function Definition

File Name: DC@F47

Description: Built-In Functions Definition file.

Normal Library: <<ppplib>>

Logical Views

Name	Use	Keys/Description/Comments
DC@F47V1	Read	F47BIF (unique).
DC@F47V2	Update	Same as DC@F47V1.
DC@F47V3	Read	F47IDN (unique).
DC@F47V4	Update	Same as DC@F47V3.

Record Layout

Field Name	Туре	Len	Dec	Description / Comments / Values
F47BIF	А	20		Built-In Function name. This name can be from 4 to 20 characters long if a 3GL BIF, to 18 characters long if an RDML BIF. It must only contain letters of the English alphabet (A - Z), underscores (_), or numerals. Additionally, it must start with the string "UD_" (User Defined for your own use) or "OV_" (Other Vendor) as described in the Naming Conventions.
F47IDN	Р	3	0	Unique number / identifier. Refer to the relevant <i>Naming Conventions</i> for recommended ranges of numbers to use. Note: If you are developing BIFs in the 601- 999 range, you must obtain from LANSA Technical Support

			(lansasupport@lansa.com.au) the next available number or a range of numbers for your use.
F47DES	А	40	A brief description of the Built-In Function in up to 40 characters.
F47COE	A	1	C or E. This determines whether the Built-In Function is C - Called or E - Executed within a function. In other words, whether this Built-In Function is a stand alone program or whether it will be part of the inline code of a LANSA function.
			The use of C (called) is recommended, because of longer term portability considerations.
			However, E (internally executed) subroutines generally offer better performance.
F47PGM	А	8	The name of a program to be called or the name of the RPG subroutine that is to be executed.
			Refer to the relevant <i>Naming Conventions</i> for notes regarding this name.
F47SLR	А	1	N if the function is *HEAVYUSAGE. Y if the function is *LIGHTUSAGE.
			This option is only a consideration when: - you are calling a program (type C)
			or - the language you are using allows the program to "stay active" between calls (e.g.: RPG or COBOL) and you intend to make use of this capability.
			If these situations apply, then set this flag to N, otherwise set it to Y.
			If this flag is set to N, then a special call will be placed to your program during termination of the RDML function indicating that your program should also terminate.

F47ARG	Р	3	0	The number of arguments (BIF_ARGnn where nn is between 0 and 20) that are passed to the Built-In Function the WITH_ARGS parameter of the LANSA USE command.
F47RET	Р	3	0	The number of returned values (BIF_RETnn where nn is between 0 and 20) that are passed by the Built-In Function in the TO_GET parameter of the LANSA USE command.

DC@F48 - Built-In Function Argument/Return Value Definition

File Name: DC@F48

Description: Built-In Functions Definition file.

Normal Library: <<ppplib>>

Logical Views

Name	Use	Keys/Description/Comments
DC@F48V1	Read	F48BIF, F48AOR, F48SEQ (unique).
DC@F48V2	Update	Same as DC@F48V1.
DC@F48V3	Read	F48BIF, F48PNO (unique).
DC@F48V4	Update	Same as DC@F48V3.

Record Layout

Field Name	Туре	Len	Dec	Description / Comments / Values
F48BIF	A	20		Built-In Function name. This is exactly the same name that is used to define the Built-In Function in file DC@F47.
F48AOR	A	3		RET or ARG. Specifies whether this parameter is an argument or return value. Arguments are identified with the value ARG, and return values are identified with the value RET.
F48SEQ	Р	3	0	The sequence of this argument or return value in relation to the other arguments or return values.
F48PNO	Р	3	0	Sequence as a CALL parm. The relative number of this argument or return value. For example, the 2nd return value could be the 8th parameter, so it should be number 8.
1	1	1	1	
--------	---	----	---	---
F48IDN	A	1		A unique identifier number/character for the argument or return value. Acceptable values are A - Z, 0 - 9. The first parameter would be identified as A, the second as B, etc.
F48DES	A	40		A brief description of this argument or return value in up to 40 characters. If the argument or return value is a field, its description. If the argument or return value is a working list, the description of its counter field if it exists.
F48ROO	A	1		R (Required) or O (Optional). R specifies that an argument or return value is required (i.e. must be specified). O indicates optional (i.e. can be left out). A default value must be specified for an Optional argument.
				R if the argument or return value is a - working list - a field without a default value.
				O if the argument or return value is a field with a default value.
				Note: A Required value cannot follow an Optional value.
F48TYP	Α	1		Parameter Type
				A (Alpha), N (Numeric) or L (List). This parameter is used to determine the type of this argument or return value. A if the argument or return value is a field of type *CHAR. N if argument or return value is a field of type *DEC.
				When L is specified, the argument or return value is a working list. When L is specified, the argument or return value used in the LANSA function must have been a working List. To use a list (L) as an

				external routine (type C). Use of lists with internal routines is not supported. N if argument or return value is a field of type REFFIELD to a signed field. Note for Built-In Function for Windows:
				Alpha, Char and String are all classed as String types and are valid arguments of type 'A'.
				Packed, Signed, Float and Integer are classed as Numbers and so are valid for arguments of type 'N'.
				All other types like Date, Datetime, BLOB and CLOB are classed as there own type and thus are not valid for either a type 'A', or type 'N'. To use these field types they must be coerced into a field type that is valid for the argument type. E.g. a Datetime can be coerced into a Char field in order to be passed as an argument of type 'A'. There is also type 'X', which means any LANSA field type, including Datetime, etc.
F48MLN	Р	11*	0	The minimum acceptable length of the field being passed from the LANSA function to the Built In Function
				1 if the argument or return value is a field.
F48XLN	Р	11*	0	The maximum acceptable length of the field being passed from the LANSA function to the Built-In Function.
				Field length if the argument or return value is a field.
				Note for Built-In Function for Windows:
				Arguments and return values can 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.
F48MDP	Р	5*	0	The minimum acceptable number of decimals that the field being passed from the LANSA function to the Built-In Function can contain. 0 if the argument or return value is a working list. 0 if the argument or return value is a field of type *CHAR. 0 if the argument or return value is a field with decimal places less than length. 1 if the argument or return value is a field with decimal places equal to length.
F48XDP	Р	5*	0	The maximum acceptable number of decimals that the field being passed from the LANSA function to the Built-In Function can contain. 0 if the argument or return value is a working list. 0 if the argument or return value is a field of type *CHAR. # of decimal places if the argument or return value is a field.
F48PLN	Р	11*	0	Pass Length: the length that the argument or return value will be passed/returned to/from the Built-In Function. Field length if the argument or return value is a field.

F48PDP	Ρ	5*	0	The number of decimals that the argument or return value will be passed/returned to/from the Built-In Function. 0 if the argument or return value is a working list. 0 if the argument or return value is a field of type *CHAR. # of decimal places if the argument or return value is a field.
F48DFT	A	10		Default value. The value to be given to this argument if one is not specified. The value can only be a numeric literal (e.g. 2, 3, 4), an alphanumeric literal in quotes (e.g.: 'A', 'IBM') or *BLANKS. *BLANK if the argument or return value is a working list. Default value if the argument or return value is a field. Note: This parameter only applies to arguments.

***Note:** The increased length of the various fields on this file apply from LANSA V11.0 only.

19.3.7 RDML Built-In Function Tips and Techniques

Initial Creation of an RDML Built-In Function

It is recommended that the RDML commands of an RDML Built-In Function be in the following sequence:

FUNCTION OPTIONS(*DIRECT ... *BUILTIN ...)

followed optionally by the override BIF name and description:

DEFINE FIELD(#BIF_NAME) TYPE(*CHAR) LENGTH(20) DESC(<BIF definition of the section of the sectio

DEFINE/DEF_LIST #BIF_ARGnn

followed optionally by the required and then optional return values:

DEFINE/DEF_LIST #BIF_RETnn

Note that a working list argument or return value is always required.

It is also recommended that the naming convention for working list argument counters be BIF_ALCnn and for working list return value counters be BIF_RLCnn

Note that these recommendations are followed in the function skeleton generated from the application template BBRDMLBIF supplied by LANSA.

Return Value Working Lists

Return value working lists are not cleared automatically as a result of either specifying them in a USE of an RDML Built-In Function or on entry to an RDML Built-In Function.

It is the responsibility of the RDML Built-In Function provider to either explicitly execute CLR_LIST commands against return value working lists in the RDML Built-In Function to document that entries are added on the end of the return value working list.

This behavior allows the RDML Built-In Function to be more flexible with regard to return value working lists.

Interactive Commands

It would be extremely unusual to code DISPLAY, POP_UP or REQUEST commands in an RDML BIF function, unless it was documented that the RDML BIF is to run only in an interactive environment and appropriate checking for such an environment were coded on entry to the RDML BIF. This would affect the portability of the RDML BIF.

Termination of RDML Built-In Function

An RDML Built-In Function that is *LIGHTUSAGE will terminate after every USE of it.

An RDML Built-In Function that is *HEAVYUSAGE will not terminate after every USE of it. Instead, the USEing function controls when it ends. If the using function is *HEAVYUSAGE it will not terminate until the program is shutdown, e.g. EXIT, but RDML commands are not executed when shutting down, so the BIF shutdown logic will not be executed. If the using function is *LIGHTUSAGE, there will be a termination call made to the BIF. Such an RDML Built-In Function can have its evaluation logic conditioned by testing that the system variable *BIF_SHUTDOWN is not 'Y'. Any special shutdown logic can be conditioned by testing that *BIF_SHUTDOWN is 'Y'.

Any RDML Built-In Function can refer to the system variable *BIF_SHUTDOWN. For RDML Built-In Functions that are *LIGHTUSAGE, its value will always be 'N', and consequently any special shutdown logic conditioned by its value being 'Y' will never be executed.

When an RDML Built-In Function is changed from *LIGHTUSAGE to *HEAVYUSAGE, or vice versa, all functions that use the RDML Built-In Function should be re-compiled if there is shutdown logic in the RDML Built-In Function.

These last three points are important. The execution of the shutdown logic is dependent not just on whether the RDML Built-In Function is marked as light or heavy usage. It also depends on whether the USEing function is light or heavy usage. The recommendation is to specify *LIGHTUSAGE in the RDML Built-In Function and to not have any logic in the shutdown logic. This means that the whole function becomes conditioned on *BIF_SHUTDOWN being 'N'. If the RDML Built-In Function is then changed to *HEAVYUSAGE, the USEing functions do not need to be re-compiled. Note that the impact of field values retaining their value between invocations would need to be investigated. The best solution to this is to always use explicit CHANGE/ASSIGN commands to set the initial value of all fields used in the function.

Platform Differences when Shutting down *HEAVYUSAGE Built-In Functions

There are differences when RPG functions as opposed to C/C++ functions shutdown *HEAVYUSAGE functions. (RPG is generated for RDML functions on IBM i. C/C++ is generated for RDMLX functions on IBM i, and both RDML and RDMLX functions on non-IBM i.)

An RPG function, when shutting down, will shutdown all *HEAVYUSAGE functions it refers to.

A C/C++ function, when shutting down, will shutdown only those *HEAVYUSAGE functions it has used while executing.

The corollary of this is that the shutdown code in the *HEAVYUSAGE Built-In Function must understand that it may be called when it hasn't been used.

An RPG function that is *HEAVYUSAGE will not get called to shutdown on IBM i unless a *LIGHTUSAGE functions uses it. Thus when the application is run again, it will still have the same state as the applications previous execution, unless Reclaim Resources has been run (RCLRSC) which will clear the state but not actually run the shutdown code.

A C/C++ Function that is *HEAVYUSAGE will get called to shutdown even when the application is terminating.

State Retention in *HEAVYUSAGE Built-In Functions

The state of fields used in *HEAVYUSAGE functions should not be relied upon due to the termination behaviour described above. For example, incrementing a value each time the RDML Built-In Function is executed. A *HEAVYUSAGE RDML Built-In Function can be called by *HEAVYUSAGE Functions and accumulate state. But as soon as a *LIGHTUSAGE Function is used, the *HEAVYUSAGE RDML Built-In Function will be shutdown and thus lose its state.

If you need to retain state, use a data area or system variable.

Optional Arguments and Return Values

It would be usual to condition references to optional return values and their derivation by testing the value of the system variable *BIF_RETCOUNT. In particular where deriving an optional return value is expensive in terms of machine resources, this would be more efficient.

Optional arguments, if not passed, will have their default value. References to optional arguments can be conditioned by testing the value of the system variable *BIF_ARGCOUNT. In particular, if the default value is not distinguishable from a passed value, but processing still needs to be different, this would be necessary.

Optional arguments and return values can be referred to in the RDML Built-In Function whether or not they have been coded in the USE command of the USEing function.

Access to an RDML BIF Function

Functions are partition specific. Built-In Function definitions are LANSA system wide. If an RDML Built-In Function has to be made available to a partition other than that where it has been compiled, it must be accessible in the library list for that other job. As with 3GL BIFS, this is the responsibility of the LANSA administrator or provider of the BIF.

Replacing 3GL BIFs with RDML BIFS

If, in the interests of portability, a 3GL BIF is replaced by an RDML BIF, all functions that USE that BIF will need to be recompiled, because of differences in the underlying architecture between 3GL and RDML BIFs

Adding Optional Arguments and Return Values

New optional arguments and return values can be added to RDML Built-In Functions without having to recompile all functions that already USE the RDML BIF. To do this the new optional arguments or return values are added after all other arguments and return values. Obviously functions that take advantage of the new optional arguments or return values will need to be recompiled. However, if the Built-In Function is **enabled for RDMLX** and it is used from any RDML function, then you **cannot** add an extra parameter, or in fact change anything to do with the number and order of arguments and return values, **unless** you recompile the RDML functions that USE this RDMLX BIF.

All other combinations are OK. That is:

- RDMLX functions to RDMLX Built-In Functions
- RDMLX functions to RDML Built-In Functions
- RDML functions to RDML Built-In Functions.

RDML BIFS in a Client / Server Environment

An RDML BIF can be used to redirect execution from the client to the server without the remainder of the application being concerned with the 'mechanics' of it.

19.3.8 Example - Define a BIF as an RDML Function

These are the steps involved in creating an RDML Built-In Function as a function. The example shown is very simple, but should give an insight into how an RDML Built-In Function is plugged into a LANSA system.

SCENARIO: This is a User Defined RDML Built-In Function. The Built-In Function is passed two arguments: a working list of packed 11,2 numbers and optionally whether to return those that are above or below the average of the numbers, defaulting to be above. The Built-In Function will return a working list of the packed 11,2 numbers as per the criteria and optionally the average value as a packed 30,9 number. The RDML BIF will be used extensively in a new application, so will be coded not to terminate between USEs.

The BIF name will be UD_GET_HI_LO_AVGE.

The BIF description will be 'Return numbers over or under the average'.

The RDML Built-In Function is generated from template BBRDMLBIF.

The argument and return value working lists are then customized from the prototype generated by the template.

The actual evaluation logic is then added.

******	Beginning of RDML commands *********

******	Copyright: (C) The LANSA Group , 19 98
******	Process: CR4587SMPL
*******	Function: CR4587S
*******	Type: Built_In Function
*******	Created by: KEVIN
*******	Created on: 02/09/98 at 19:46:50
*******	Description: Sample RDML BIF Function
******	=======================================
FUNCTION ILTIN	OPTIONS(*DIRECT *NOMESSAGES *HEAVYUSAGE *ML N)

********* ****	Special field to name the Built-In Function
DEFINE I mbers _AVG	FIELD(#BIF_NAME) TYPE(*CHAR) LENGTH(20) DESC('Retu over or under the average') DEFAULT('UD_GET_HI_LO 5')

********** Built-In Function Arguments ******** ______ _____ ******* Argument 01, Supplied numbers list FIELD(#BIF_ALC01) TYPE(*DEC) LENGTH(7) DECIMALS(0) DEFINE ('Supplied numbers list') FIELD(#NUMBER) TYPE(*DEC) LENGTH(11) DECIMALS(2) DEFINE Number') DEF_LIST NAME(#BIF_ARG01) FIELDS((#NUMBER)) COUNTER(#BIF TYPE(*WORKING) ENTRYS(9999) ******* Argument 02, Over or under numbers required (O or U) FIELD(#BIF_ARG02) TYPE(*CHAR) LENGTH(1) DESC('Over DEFINE nder numbers required (O or U)') DEFAULT('O') *********** Working fields, lists and groups ******** ______ FIELD(#TOTAL) TYPE(*DEC) LENGTH(30) DECIMALS(2) DI DEFINE otal') ******** Return Value 01, Over or under average numbers list DEFINE FIELD(#BIF_RLC01) TYPE(*DEC) LENGTH(7) DECIMALS(0) ('Over or under average numbers list') DEF_LIST NAME(#BIF_RET01) FIELDS((#NUMBER)) COUNTER(#BIF_ TYPE(*WORKING) ENTRYS(9998) ****** Return Value 02, Average of supplied numbers FIELD(#BIF RET02) TYPE(*DEC) LENGTH(30) DECIMALS(§ DEFINE C('Average of supplied numbers') DEFAULT(*ZERO) ******** Function Mainline : CR4587S ******** This is an evaluation call ****** ******* COND('*BIF SHUTDOWN *NE Y') IF ******** calculate the average FIELD(#TOTAL) TO(0) CHANGE SELECTLIST NAMED(#BIF ARG01)

CHANGE FIELD(#TOTAL) TO('#TOTAL + #NUMBER') ENDSELECT

CHANGE FIELD(#BIF_RET02) TO('#TOTAL / #BIF_ALC01') ROUND_UES)

```
CLR_LIST NAMED(#BIF_RET01)
SELECTLIST NAMED(#BIF_ARG01) WHERE('((#BIF_ARG02 *EQ O) *A
BER *GT #BIF_RET02)) *OR ((#BIF_ARG02 *EQ U) *AND
(#NUMBER *LT #BIF_RET02))')
ADD_ENTRY TO_LIST(#BIF_RET01)
ENDSELECT
********* This is a shutdown call
******
```

ELSE

********* No shutdown logic

ENDIF

********* Return control to the invoker

RETURN

********** End of RDML commands *********

Note that:

- The return value list is explicitly cleared prior to any entries being added.
- The optional argument and return value can be referred to even though they may not have been coded in the USE statement. It would be more usual to condition references to any optional arguments or return values by testing the value of *BIF_ARGCOUNT and *BIF_RETCOUNT respectively. There is no extra processing to derive the optional return value. It saves defining another work field.
- Because the function is heavyusage, its evaluation logic is conditioned by ensuring that it is not a shutdown call. Also the accumulation field is reset to 0 every time. For this RDML BIF there is no special shutdown logic.

19.3.9 Package and Deploy RDML BIFs

Package RDML Built-In Functions for Distribution (IBM i only) Install Other Vendor RDML Built-In Functions (IBM i only) Package Built-In Functions for Deployment (Windows/Linux)

Package RDML Built-In Functions for Distribution (IBM i only)

Create duplicate objects of any Built-In Function objects from your partition module library to library QTEMP. For example:

```
CRTDUPOBJ OBJ(@OV_nnn*)
FROMLIB(<partition module library>)
OBJTYPE(*PGM)
TOLIB(QTEMP)
```

where

<partition module library> is the module library for the partition where the RDML Built-In Functions are compiled.

Copy relevant records from DC@F47 and DC@F48 to temporary versions of these files in QTEMP. For example:

```
CPYF FROMFILE(<pgmlib>/DC@F47)
TOFILE(QTEMP/DC@F47)
MBROPT(*ADD)
CRTFILE(*YES)
INCREL((*IF F47BIF *EQ OV_XXXXXXXXXXXX)
(*OR F47BIF *EQ OV_YYYYYYYYYY))
```

```
CPYF FROMFILE(<pgmlib>/DC@F48)
TOFILE(QTEMP/DC@F48)
MBROPT(*ADD)
CRTFILE(*YES) FMTOPT(*MAP *DROP)
INCREL((*IF F47BIF *EQ OV_XXXXXXXXXXX)
(*OR F47BIF *EQ OV_YYYYYYYYYY))
```

where

<pgmlib> is the LANSA Program Library Save QTEMP to the save media.

Install Other Vendor RDML Built-In Functions (IBM i only)

Restore objects from save media to QTEMP.

Create duplicate objects of any Built-In Function objects from library QTEMP to your Built-In Function library. For example:

```
CRTDUPOBJ OBJ(@OV_nnn*)
FROMLIB(QTEMP)
OBJTYPE(*PGM)
TOLIB(<bif pgms>)
```

where

<bif pgms> = the library where the Built-In Function objects are stored
Copy all records from DC@F47 and DC@F48 from QTEMP to the LANSA
Program Library version of these files.

For example:

```
CPYF FROMFILE(QTEMP/DC@F47)
TOFILE(<pgmlib>/DC@F47)
MBROPT(*ADD)
CPYF FROMFILE(QTEMP/DC@F48)
TOFILE(<pgmlib>/DC@F48)
MBROPT(*ADD)
```

where

<pgmlib> = LANSA Program Library

Package Built-In Functions for Deployment (Windows/Linux)

You need to be familiar with the Deployment Tool shipped as part of Visual LANSA in order to be able to package objects from your Visual LANSA environment.

To include the User Defined Built-In functions in your package, you will need to select them from the *Objects* Tab from your package definition.

- To select a 3GL BIF, select it from the *Built-In Functions* node.
- To select an RDML BIF, select it from the *Processes/Functions* node.

• To deploy the BIF definitions (LX_F47 and LX_F48 files), for example, when deploying to a development environment, you must select the *Deploy System/Partition Definitions* option in the Deployment Tool's *Package Settings*. Enabling this option will add the BIF definitions to the deployment package.

19.4 Built-In Function Sample Form

Following are sample forms which you could use when designing Built-In Functions. They are intended as an aid for entering the DC@F47 and DC@F48 file definitions.

BIF Definition (as per DC@F47 file layout)	
BIF Name:	
Unique Identifier:	
Description:	
Call or Execute: (C or E)	
Program Name:	
Terminates between calls: (Y or N)	
Number of arguments:	
Number of return values:	
BIF Arguments (as per DC@F48 file layout)	
BIF Name:	
Parameter Type ARG	
Parameter Sequence:	
Parameter Number:	
Parameter Identifier:	
Description:	
Required / Optional: (R or O)	
Parameter Type: (A, N or L)	
Minimum Length:	
Maximum Length:	

Minimum Decimal:		
Maximum Decimal:		
Pass Length:		
Pass Decimal:		
Default:		
BIF Return Values	(as j	per DC@F48 file layout)
BIF Name:		
Parameter Type:	RET	
Parameter Sequence		
Parameter Number:		
Parameter Identifier:		
Description:		
Required / Optional:		(R or O)
Parameter Type:		(A, N or L)
Minimum Length:		
Maximum Length:		
Minimum Decimal:		
Maximum Decimal:		
Pass Length:		
Pass Decimal:		

19.5 Built-In Function Skeletons

The application template BBRDMLBIF will generate the function skeleton for an RDML Built-In Function. You may wish to review 19.3.7 RDML Built-In Function Tips and Techniques prior to executing the template.

- 19.5.1 RPG/400 and ILE Skeleton Program
- 19.5.2 Control Language (CL) Skeleton
- 19.5.3 Subroutine Skeleton
- 19.5.4 C Skeleton Program

19.5.1 RPG/400 and ILE Skeleton Program

The sample source code is supplied in the source file DC@F28 in your LANSA data library, which is usually called DC@DTALIB.

The RPG versions are:

- Built-In Function Skeleton is BI@P001RPG
- The /copy version is DC@ISPEC
- The /copy version is PR@ISPEC
- The /copy version is DC@ESPEC
- The /copy version is PR@ESPEC

The RPG ILE version are:

- Built-In Function Skeleton is BI@P001RIL
- The /copy version is DC@DSPEC
- The /copy version is PR@DSPEC

19.5.2 Control Language (CL) Skeleton

The following sample source code can be found as member BI@P001CLP in source file DC@F28 in your LANSA data library, which is usually called DC@DTALIB.

/* /*	Standard Built-In Function Provider Disclaimer */
/*=== /*	+
Pro	vider Name : XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
The	SIF facility is a very powerful and very open interface to +
LAI	
or pr total has port Alth defin any	<pre>cchase other vendor supplied Built-In Functions you are + / responsible and totally liable for any effect whatsoever it + n the integrity, usability, security, maintainability or + oility of any LANSA generated application on your system(s). + + ugh your product vendor may assist you with Built-In Function + tions and maintenance, they are not responsible or liable in + yay for its function. +</pre>
/*=== /*	Basic Program Details */
/*=== /*	+
Cop	right :XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
D	+
Pro	ram Name : XXXXXXX +
Bui	in Name : XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
-	+
Dat	written : DD/DD/DD +
Aut	ors Name : XXXXXXXXXXXXXX +
41	+



DCL VAR(&DC@IDS) TYPE(*CHAR) LEN(1024)

DCL DCL	VAR(&DC@EDS) TYPE(*CHAR) LEN(1024) VAR(&PR@IDS) TYPE(*CHAR) LEN(1024)
DCL	VAR(&\$PGMNM) TYPE(*CHAR) LEN(10)
DCL	VAR(&PR@PGM) TYPE(*CHAR) LEN(10)
DCL	VAR(&PR@B@N) TYPE(*CHAR) LEN(3)
DCL	VAR(&PR@B@A) TYPE(*CHAR) LEN(1)
DCL	VAR(&DC@RET) TYPE(*CHAR) LEN(1)
DCL	VAR(&DC@MID) TYPE(*CHAR) LEN(7)
DCL	VAR(&DC@MVR) TYPE(*CHAR) LEN(132)
DCL	VAR(&@COPYR) TYPE(*CHAR) LEN(80) VALUE('(C) +
	COPYRIGHT < <your +<="" name="" organization's="" td=""></your>
	here>>, 1991. ALL RIGHTS RESERVED')

/*========		=======================================
/*	Global Error Handler	*/
/*========		=======================================

MONMSG MSGID(CPF0000 MCH0000) EXEC(GOTO ARGERR)

/*	
/*	Program Mainline */
/*	

/* Set up some fields and map some from the data structures */

&\$PGMNM	'??????????
&PR@PGM	%SST(&PR@IDS 424 10)
&PR@B@N	%SST(&PR@IDS 157 3)
&PR@B@A	%SST(&PR@IDS 160 1)
	&\$PGMNM &PR@PGM &PR@B@N &PR@B@A

/* Initially assume that a "good" return will be the result */

CHGVAR &DC@RET 'Y'

/* Perform the required evaluation / action here */

/* End of program processing logic */

ENDPGM: CHGVAR %SST(&DC@EDS 38 1) &DC@RET CHGVAR %SST(&DC@EDS 39 7) &DC@MID CHGVAR %SST(&DC@EDS 46 132) &DC@MVR RETURN

ARGERR:

/* Route messages any additional messages back to the caller */

CALL PGM(DC@P9007) PARM(&\$PGMNM &PR@PGM) MONMSG (CPF0000 MCH0000)

/* Set up a "bad" return code

*/

CHGVAR &DC@RET 'N' MONMSG (CPF0000 MCH0000)

/* Return control to the calling program */

GOTO ENDPGM

CHGVAR VAR(%SST(&@COPYR 80 1)) VALUE(' ') ENDPGM

19.5.3 Subroutine Skeleton

C* XXXXXX : << Description goes here >> С XXXXXX BEGSR C* C* Initially assume that no error will occur С* MOVEL'Y' DC@RET С C* C* *** Code Builtin Logic here *** C^* С **ENDSR**

19.5.4 C Skeleton Program

/* _________ */ /* ======== 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 powefull 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. */ /* */ /* */ /* */ /* Source File : U BIFnnn.C */ /* Entry Point Name : U_Builtin_nnn /* Linked DLL Name : U BIFnnn.DLL */ /* Shared Object Name (Linux): u_bifnnn.O */ /* OS/Dependencies : Yes/No */ /* */ /* Amendment History : */ /* */ /* Task Id Date Description */ */ /* ====== ==== ============

```
#define U_BIF_FUNCTION
                           U BuiltIn nnn
#define U_BIF_FUNCTION_NAME "U_BuiltIn_nnn"
#define U_BIF_DESCRIPTION "This is a description of this builtin"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "x glodef.h"
#include "x_glousr.h"
#ifdef X_OPERATING_SYSTEM_WIN
#include <windows.h>
#endif
#include "x funstr.h"
#include "x_funpro.h"
#include "x_bif000.h"
/*_____
                                             _____
/*
                                   */
/* Arguments
                       - Standard X IDS system definition
            :pX Ids
                                                       */
                   - Standard X PRO process definition */
/*
         pX Pro
/*
         pX Fun
                    - Standard X FUN function definition */
/*
         pX Bif
                   - Standard X BIF builtin definition
                                                  */
/*
         X Fld[]
                   - Standard X FLD field definitions
                                                  */
/*
         X List[]
                   - Standard X LIST list definitions
                                                 */
         sInCount - Number of arguments passed in
/*
                                                  */
/*
         sInVec[] - Vectors of arguments
                                              */
/*
         sRetCount - Number of return values
                                                */
/*
         sRetVec[] - Vectors of return values
                                              */
/*
                                    */
```



20. Application Templates

Application templates in LANSA are provided to enable RDML programs to be generated automatically by a "question and answer" session.

An application template consists of Application Template Commands and RDML commands.

The application template commands control how, when and what RDML commands are generated. They usually do this by asking the user a question, then by using the answer, generate different RDML commands.

The application template may be used as a "skeleton" or "model" for the generation of a complete RDML program or to generate a commonly used section of RDML code or logic.

"End user" access to application templates is supported when creating a new RDML function. An end user may select an application template that will generate a complete RDML program by answering the questions defined in the template.

Programmer access to application templates is supported when both creating a new function, and directly from the RDML editor. The programmer may choose many different application templates to automatically construct different parts of the RDML program they are creating or maintaining.

Some application templates are shipped with the LANSA product.

You can modify templates that are shipped with the product to exactly match your installation standards.

You can create your own application templates for subsequent use by your programmers or end users.

This guide does not deal with:

- How the application template commands are constructed and used.
- How an application template is invoked and then used by a programmer or an end user to generate an RDML program.

Refer to the LANSA for iSeries User Guide for more details of this area.

• For a list of templates shipped with LANSA, refer to Template Commands and Variables in the LANSA Technical Reference Guide.

20.1 Who Should Write Application Templates?

Writing and maintaining application templates should only be attempted by very experienced programmers who have extensive knowledge of both Application Template Commands and RDML Commands.

A suggested way to familiarize yourself with application templates is to print one or more of the LANSA supplied application templates and work your way through them. Sample application template programs are also provided at the end of this section.

The execution of certain Application Template Commands may result in error/warning messages being displayed. Refer to the Technical Reference Guide for more details.

If you cannot understand what/how these application template commands work, then DO NOT ATTEMPT to create or modify any application templates.

Before any Application Template is made available for public use, it should be EXTENSIVELY TESTED for all possible situations. This means generating a number of RDML programs and making sure they compile and then execute with the desired results. The importance of this step cannot be over emphasized.

Expect an application template, including detailed HELP panels, to take you at least one full day to write and test.

20.1.1 Application template programming

- Is a job for very experienced programmers.
- Is a job for someone with detailed knowledge of your site's standards.
- Is a job for someone who understands the concept of transaction "skeleton" or "models".
- Requires an eye for detail.

20.1.2 Responsibility

Remember that when application template code is written or modified, its resulting RDML code is entirely **your responsibility.** Although your product vendor will assist you in writing and amending application templates, the vendor is not responsible for whatever is generated by the application template, nor is the vendor liable in any way to correct any problems it causes.

20.2 The Templates Shipped with LANSA

A number of Application Templates are shipped with LANSA. You should note the following about these templates before attempting to use them:

- They reflect a reasonable cross section of the types of transactions you would most likely wish to perform.
- They are generalized, and thus are unlikely to conform to your sites standards. In many respects they are shipped as examples of different types of application templates and what templates can be used for. They should be copied and modified/extended/ customized to meet your site's requirements.
- They are designed to work with normalized database structures and database designs that would be considered conventional in "mainstream" commercial data processing. Thus some types of data structures (e.g. un-normalized design or large numbers of fields per file) may require the construction of very specialized application templates.
- Because of the way Application Templates interact with the user, no warranty can be expressed or implied at the RDML code level for code generated by any shipped Application Template. It is the responsibility of the user to test the resulting RDML code just like any manually coded RDML program.
- New "shipped" Application Templates are being created all the time. If you wish to import the latest set of templates then contact your product vendor.
- If you have some ideas/examples for new Application Templates, please don't hesitate to send the details to your product vendor.

20.3 Application Template Commands

All "special" application template commands are set up using the command definition facilities of the IBM i operating system.

All such commands conform to the normal command conventions. For instance, positional or keyword parameter specification can be used, lists must be enclosed in brackets, etc..

If you are unfamiliar with the method of specifying IBM i operating system commands refer to the appropriate IBM supplied manual for more information.

All application template commands in the following sections include a "syntax diagram".

This method of documenting a command and its associated parameters is similar to that used in the IBM Control Language Reference Manuals for the IBM i. Refer to the relevant manual for a full explanation of "syntax diagrams".

All "special" application template commands begin with "@@". For example, "@@QUESTION" is an application template command which instructs LANSA to ask the user a question and receive an answer.

An application template consists of these "special" commands and RDML code. The commands control the generation of the RDML code. For full details of all LANSA RDML commands, refer to the preceding sections of this guide.

The application template commands and RDML code can also make use of "special" variables. These variables will be substituted in the RDML code by LANSA before being added to the generated RDML program.

For a list of the current application template commands and the special variables, refer to Application Templates in the *Visual LANSA Developers Guide*.