# Visual LANSA Developer Guide

- Getting Started with LANSA Development
- Creating LANSA Applications
- Creating Fields
- Creating Application Databases
- Creating Applications Using Functions
- Creating Applications with Components
- ActiveX Controls
- .NET Components
- Application Templates
- Appendix A. System Variable Evaluation Programs
- Appendix B. Domino Integration Examples

Edition Date October 27, 2014 © LANSA

## **1. Getting Started with LANSA Development**

If you are a new LANSA developer, you should begin by reviewing the following topics:

- 1.1 Developer Roadmap
- 1.2 LANSA Architecture
- 1.3 LANSA Repository
- 1.4 LANSA RDML
- **1.5 Developer Resources**

#### Also See

Development Environment Administration in the Administrator Guide

# 1.1 Developer Roadmap

LANSA developers may be involved in building many different types of applications. The purpose of the Developer Roadmap is to provide a perspective for LANSA development based on the focus of your project.

- 1.1.1 IBM i Development
- 1.1.2 Windows/Linux Development
- 1.1.3 Web Development
- 1.1.4 LANSA Integrator Development
- 1.1.5 LANSA Open Development
- 1.1.6 LANSA Client Development
- $\uparrow$  1. Getting Started with LANSA Development

### 1.1.1 IBM i Development

If you are developing applications for the IBM i, you can use the Visual LANSA or the LANSA for i development environment. All of your IBM i development needs can be met using the Visual LANSA development environment, and the Visual LANSA development environment is recommended for LANSA developers. The LANSA for i development cannot be used with RDMLX Partitions.

Many IBM i-based applications will use the Universal Model for applications. If you are unsure of the type of application you require, review Creating LANSA Applications.

Your first steps in the project will be Creating Fields and Creating Application Databases.

It is important to review how Application Templates can improve your productivity before you begin Creating Applications using Functions.

If you are developing Internet applications for IBM i, you may also be doing 1.1.3 Web Development or 1.1.4 LANSA Integrator Development.

If your are other PC based tools (such as Visual Basic or Delphi) to develop client/server application using an IBM i database, review 1.1.5 LANSA Open Development.

### Also See

LANSA for i User Guide

### **1.1.2 Windows/Linux Development**

If you are developing graphical applications for Windows, Linux or other platforms, you will use the Visual LANSA development environment and then deploy your applications to the desired platform.

If you are unsure of the type of application you require, review Creating LANSA Applications.

Your first steps in the project will be Creating Fields and Creating Application Databases.

It is important to review how Application Templates can improve your productivity before you begin Creating Application using Forms.

If you are developing Internet applications, you may also be doing 1.1.3 Web Development or 1.1.4 LANSA Integrator Development.

#### Also See

Deploying Visual LANSA Applications on Linux Guide Visual LANSA Deployment Tool Guide

## 1.1.3 Web Development

If you are developing Web applications using LANSA, you have a number of development choices. LANSA for the Web supports both Web Function applications and component-based Web Application Modules. You will use the Visual LANSA development environment and then deploy your applications to the desired platform.

Your first steps in the project will be Creating Fields and Creating Application Databases that form the basis of your Web applications.

If you are developing a Web application, you should also refer to the: Web Application Modules Guide.

**1.1.4 LANSA Integrator Development** can also be used for constructing applications using the Internet to connect businesses and their applications (A2A and B2B).

#### Also See

Web Administration Guide

## **1.1.4 LANSA Integrator Development**

LANSA Integrator is used for integration of Application-to-Application (A2A) and Business-to-Business (B2B) transactions through XML and Java Services. LANSA Integrator allows bi-directional XML and other data formats to be exchanged between businesses across different platforms.

Your first steps in an Integrator project will be Creating Fields and Creating Application Databases that form the basis of your transactions.

Once the databases are defined, you can create your Integrator applications using either functions or components.

### Also See

LANSA Integrator Guide

## 1.1.5 LANSA Open Development

If you are using the LANSA Open to build client/server applications, your focus will be on the development of the LANSA Repository to support your client applications. LANSA Open allows other Windows development tools to share and reuse LANSA's Repository and RDML technologies.

It is very important that you understand the 1.2 LANSA Architecture and the 1.3 LANSA Repository when building client/server application with LANSA Open.

Your first steps will be Creating Application Databases that will typically be done by loading your existing files into the LANSA Repository.

Of particular importance, will be File Rules and Triggers Development, Virtual Field Development and possibly Predetermined Join Field Development in the repository.

Once you have completed the repository development, you can begin creating your client/server applications with LANSA Open.

#### Also See

LANSA Open Guide

## **1.1.6 LANSA Client Development**

If you are using the LANSA Client end-user query and reporting tool, your focus will be on the development of the LANSA Repository file definitions to support LANSA Client users.

Your first steps will be Creating Application Databases that will typically be done by loading your existing files into the LANSA Repository.

Of particular importance, will be Access Route Development. You will also want to include Virtual Field Development and possibly Predetermined Join Field Development in the repository.

Once you have completed the repository development, you can begin to execute queries and reports with LANSA Client.

#### Also See

LANSA Client Guide LANSA Client Administrators Guide

## 1.2 LANSA Architecture

If you understand the fundamental LANSA architecture, you may develop applications that take advantage of the architecture features. Review the following topics:

- 1.2.1 LANSA Product Architecture
- 1.2.2 Centralizing Information in A Repository
- 1.2.3 Repository and RDML
- 1.2.4 Traditional Application Architecture
- 1.2.5 Tiered Application Architecture
- 1.2.6 LANSA Application Architecture
- 1.2.7 LANSA is an Architecture
- ↑ 1. Getting Started with LANSA Development

## **1.2.1 LANSA Product Architecture**

Each of the products in the LANSA family shares a common architecture based on a Repository Technology and a High-Level Application Definition using a Fourth Generation Language (4GL). The LANSA Active Object Repository and 4GL, called 1.4 LANSA RDML (Rapid Development and Maintenance Language), work together to provide a complete development and execution solution.

The LANSA Repository stores a vast amount of information about your business application in one central location. (Refer to 1.2.2 Centralizing Information in A Repository.) This information is used to build and execute your application. The centrally defined repository improves developer productivity while reducing maintenance. For distributed network applications, the repository provides superior performance characteristics.

LANSA uses a powerful 4GL (Fourth Generation Language) to create a High-Level Application Definition of your business logic. Using a 4GL makes your application independent of the operating platform so that you can create just one application definition which can execute on multiple platforms. LANSA's RDML allows developers to more productively define procedural logic which significantly reduces maintenance.

All of the products in the LANSA family can make use of Repository Technology and the LANSA RDML.

### Also See

- 1.2.3 Repository and RDML
- 1.2 LANSA Architecture

# **1.2.2 Centralizing Information in A Repository**

Many traditional 3GL programs contain a great deal of redundant information about the application. Information such as a description of a field, its defaults and valid values may be stored in each separate program which uses a particular field. Rules such as referential file integrity or file relationships are often required. A large amount of the logic in programs is simply repeated business definitions and business validation rules.

The LANSA architecture helps developers to overcome the above problems by centralizing the information about the business application in the LANSA Active Object Repository. It removes redundant code from the programming environment. It ensures that common business logic is shared and reused by storing it just once in a central location - the Repository. And because the LANSA architecture is open, the information in the LANSA Repository can also be used by other development tools.

Development with 1.4 LANSA RDML becomes much more productive because the developer reuses the centralized application definitions of objects in the repository. RDML programs become much smaller and simpler as they do not duplicate information that has already been centralized in the repository. The developer writes significantly less code and is able to generate applications faster by reusing the information in the repository.

The LANSA Repository simplifies maintenance because business changes can be made to the central definition in the repository instead of changing each application program. In fact, the application program may not require any recompilation at all. Impact analysis, coding, and testing are all simplified due to LANSA's ability to centralize information.

#### Also See

1.2.3 Repository and RDML ↑ 1.2 LANSA Architecture

# 1.2.3 Repository and RDML

The LANSA Repository and RDML programming language work together to create an integrated solution. You use both components to build your applications. The resulting LANSA development environment is highly productive and easy to use. Many benefits result from the combination of these two technologies.

You cannot create a LANSA application by using just the RDML programming language. You must define information in the LANSA Repository before you can begin to create your application logic. In fact, the definition of the business application in the LANSA Repository is the first step in building your application. The repository stores the information about the fields and the files that are used by the application. It stores details about the related business rules for these fields and files. It can also store components which make up your applications.

**1.4 LANSA RDML** is much more productive because it has been designed to reuse the information it knows is already defined in the repository. For example, default screen layouts can be automatically generated based on the information in the repository. Developers do not need to code business rules which have already been defined in the repository. Also, features like help text and field level prompting are fully supported by the repository and do not have to be coded into the application.

This application architecture, combining repository and 4GL technologies, has some significant advantages when deploying applications in today's computing environment which involves many different hardware and software platforms.

#### Also See

1.2.7 LANSA is an Architecture

1.2 LANSA Architecture

# **1.2.4 Traditional Application Architecture**

Application programs may be divided into three layers:

### **Presentation Logic**

• This layer handles the interface to the user allowing the user to enter information and receive messages back from the program.

### **Application or Business Processing Logic**

• This layer performs the business calculations on entered data.

### **Database or Business Rule Logic**

• This layer performs business rule validations and data transformations before information is accessed/updated in the database.

In the past, the traditional program architecture or application programming model was based on executing application programs on a single hardware platform. Under this model, the application program could combine the presentation logic, application logic and database logic into a single large program. Business applications were made up of hundreds of these large all-inone programs.

For example, traditional IBM i application programs are large RPG-based programs which are coded to handle screens, processing logic, and access to the IBM i database. These types of programs are time consuming to produce and difficult to maintain. They have limited or no portability options. There is limited code reuse and application logic is often duplicated in many places.

A traditional architecture, which forces developers to combine all layers into a single program, has serious limitations once a network or distributed computing environment is introduced. The 1.2.5 Tiered Application Architecture provides greater flexibility.

### Also See

- 1.2.6 LANSA Application Architecture
- 1.2 LANSA Architecture

# **1.2.5 Tiered Application Architecture**

Computing environments often involves multiple hardware platforms and operating systems. Application programs are distributed across these platforms. To facilitate a distributed processing model, developers introduced tiered architectures.

Tiered architectures involve the separation of the different processing segments of an application:

#### Presentation Logic (Graphical Interfaces, Etc.)

• This logic is handled by specialized client (PC-based) applications with robust graphical interfaces that simply control the presentation of information (such as a Web browser).

#### **Application or Business Processing Logic**

• This logic may reside on a client as a called function or may reside on the server as a remote procedure call but is often stored as a separate module. Application servers may support the application logic but contain no databases.

#### **Database or Business Rule Logic**

• This logic may be entered into a specific database manager so that a referential integrity check or business rule triggers are invoked by the database manager and not the application programs.

A tiered architecture provides a significantly more flexible computing model but introduces a number of new challenges. These layers in the architecture are often written in different languages with different tools. They reside on different platforms and execute under different operating systems. The final solution may still lack portability because of the platform specific requirements written into each layer.

A development environment with a properly architected solution can easily address these complexities and simplify application development. The 1.2.6 LANSA Application Architecture simplifies application development.

#### Also See

1.2.4 Traditional Application Architecture

1.2 LANSA Architecture

# **1.2.6 LANSA Application Architecture**

LANSA's repository-based 4GL solution offers the benefits of a fully tiered program architecture while insulating the developer from the complexities of the deployment environment. A single 1.4 LANSA RDML program can generate multiple languages to execute on multiple platforms. The 1.3 LANSA Repository centralizes the core business information where it can be shared and reused, and easily moved from one database to another.

It is important to remember that when you build a LANSA application you have:

#### **Platform Independent Application Architecture**

- Your application can execute on multiple platforms because it is not written for a specific platform or operating system.
- You can execute applications in a single machine environment or you can easily deploy the same applications in a heterogeneous computing environment.

### **Centralized Application Definition**

- Your data and business rules are stored once in a centralized repository so they are easily maintained.
- Using the repository, you can share and reuse key parts of your application logic and definitions.

### **Portable Database Architecture**

- The repository makes LANSA applications independent of the database manager.
- The database application architecture is fully portable across multiple database managers.
- Stored procedures or trigger programs are written in a platform independent 4GL and do not have to be recoded when moved to new platforms.

Since LANSA is an integrated solution, you are dealing with just one product and one set of development skills to meet your multiplatform host and client/server development needs. It is important to understand that 1.2.7 LANSA is an Architecture.

#### Also See

1.2.4 Traditional Application Architecture

1.2.5 Tiered Application Architecture ↑ 1.2 LANSA Architecture

### **1.2.7 LANSA is an Architecture**

LANSA is an architecture and not a software program. It is an architecture for developing applications instead of being just an application programming language or tool.

For example, with LANSA, a developer can create a single business application solution which can:

- execute as a standalone application on a PC, or PC network running Windows
- execute over the World Wide Web as an Internet or Intranet application using a browser or other device supporting Java or XML
- execute as a client/server application supporting distributed presentation, distributed logic or distributed data models from a Windows client to an IBM i or other server
- execute as a standalone 5250 host-based application on the IBM i

and this solution is:

- created using a single set of skills
- created from one set of code (a single application definition)
- created using a single family of integrated products.

LANSA is a repository-based 4GL so that your applications are properly architected for the future. An investment in a LANSA application is an investment in an application architecture which will protect your business application systems.

Once you have invested in a LANSA application, you have a wide range of deployment options for your application.

### ↑ 1.2 LANSA Architecture

## 1.3 LANSA Repository

Developers must understand a repository-based development paradigm to properly use LANSA. Review the following topics:

- 1.3.1 What is a Repository?
- 1.3.2 What is An Active Repository?
- 1.3.3 What Goes in LANSA Repository?
- 1.3.4 Repository Benefits
- $\uparrow$  1. Getting Started with LANSA Development

## 1.3.1 What is a Repository?

A repository stores information about an application in a central location. It is simply a collection of information about the application - the application's fields, files, rules, relationships, etc. Like a High-Level Application Definition, it should store the information in a non-platform specific format.

Using a repository architecture should significantly reduce application coding. Application information is defined just once in a central location instead of repeating this information wherever it is used in application programs. For example, a business rule for a field can be defined once instead of on every screen and program which uses that field.

Repositories should promote both the sharing and reuse of application logic. Sharing and reuse are very similar but not the same. Reuse generally applies to code reuse within one application. However, completely different applications can share objects or logic when they use a repository, just as different applications can share a database file. Different applications can also reuse repository-based information so it is not repeated in multiple places. It is important to note that some tools can reuse information in an application, but the information cannot be shared with a different application or tool.

Another key benefit of repositories should be reduced maintenance. Repositories simplify application maintenance because the information is easy to find and change since it is stored centrally. Testing can be much faster and easier. Changes can often be made to the repository without impacting existing coded application logic.

Finally, a repository architecture should increase developer productivity, particularly for client/server application development. Developer productivity increases are a direct result of reduced coding, reduced maintenance, reuse and sharing of application logic.

However, a repository does not guarantee these benefits to the developer. The manner in which the repository architecture is implemented is very important. Not all repositories are the same.

#### Also See

1.3.2 What is An Active Repository?1.3.3 What Goes in LANSA Repository?↑ 1.3 LANSA Repository

## 1.3.2 What is An Active Repository?

Many development tools have some form of a repository. This repository may simply be a central place to store application source code and information which is used ONLY to compile application programs. This type of repository is designed to simply allow information to be reused when recompiling the application. It is a passive repository and has limited benefits to the application developer.

However, an ACTIVE repository is used in both the building and execution of the applications. It is dynamic and it is used when the application executes. Being dynamic also means that changes made to the repository can be immediately reflected in the application without having to rebuild coded application programs.

For example, in a passive repository, the field that stores a person's age may have a text description of "Employee Age" which is stored in the repository. This description is used on a screen created for hiring a new employee or adding a new employee to the database.

However, in an active repository, the employee age field can have a validation rule added which says an employee must be at least 16 years old. This rule could be added to the repository instead of having to change validation rules in the hire an employee program. When the program executes, the new rule would be enforced by the active repository. With an active repository, help text can be added just once in a central location. This text is immediately available on all screens which have the employee age field.

An active repository technology provides real benefits to the developer because of its architecture. Active repositories introduce an execution component into the overall application program architecture. Also, a greater amount of information can be stored in the repository and this information can be shared by other applications. Application maintenance is reduced and developer's productivity is increased.

LANSA is one of the few application development tools which provides an Active Repository!

#### Also See

- 1.3.1 What is a Repository?
- 1.3.3 What Goes in LANSA Repository?
- 1.3 LANSA Repository

# 1.3.3 What Goes in LANSA Repository?

The LANSA Repository stores a vast range of information about a business application. Again, this information is stored in a non-platform specific format so that it can be used on multiple platforms. It stores information used to build and execute the application.

The LANSA Repository stores information about fields or elements in the application such as descriptions, column headings, edit codes, visualizations, default values, help text, prompt programs, etc. It acts like a data dictionary for your application. It also stores objects or components which can be used for Windows Optimized applications which are event-driven.

The LANSA Repository also stores information about the files or application database such as physical files, logical files, file relationships, file definition attributes, file validation rules, trigger programs, etc.

Further, the LANSA Repository stores procedural information or business rules about the application in the form of stored procedures, trigger functions, validation programs, system variables and more. LANSA's RDML is used for the business logic definition.

Finally, LANSA offers some special repository features such as multilingual definitions, virtual fields, predetermined join fields, and more.

Remember, once this information has been stored in the central repository, there is no need to repeat this information in any application program.

Also, the information in the LANSA Repository is reused and shared. Information can be reused by other products in the LANSA product family and by other applications and development tools. LANSA is an open architecture.

#### Also See

1.3.2 What is An Active Repository?↑ 1.3 LANSA Repository

# **1.3.4 Repository Benefits**

LANSA's Active Repository provides numerous benefits to application developers and end users of applications for both host and distributed (client/server) computing.

- The LANSA Repository centralizes information about your business application so that logic is both reused and shared (host & client/server). A single, centralized definition, which is open to other products, protects your information systems investment.
- LANSA significantly increases developer productivity and application performance (thin client) because of the reuse and sharing of the centralized application definition.
- A centralized application definition also simplifies maintenance of the application. Information is easy to locate, modify, test and deploy. Client/server applications benefit substantially from this architecture.
- LANSA offers a powerful set of development tools within the repository. LANSA's multilingual definitions, virtual fields and predetermined join fields are examples of features which simplify and accelerate development.
- The repository is easily used with existing databases so that you can extend your existing application. After loading existing files into LANSA, repository features such as virtual fields or predetermined join fields can still be added.
- The LANSA Repository is open to your choice of tool. Developers can choose a variety of development tools and still benefit from the repository architecture. Again, LANSA shares and reuses the information in the repository.

The LANSA Repository can reside on platforms such as the IBM i, Windows and Linux.

#### Also See

- 1.3.2 What is An Active Repository?
- 1.3.3 What Goes in LANSA Repository?
- 1.3 LANSA Repository

## 1.4 LANSA RDML

Building applications with a repository-based 4GL is not the same as coding in 3GLs. The following section introduces the LANSA programming language.

1.4.1 Why a 4GL / RDML?

1.4.2 RDML Structures

1.4.3 Programming Productivity

1.4.4 Where is RDML Used?

1.4.5 LANSA Application Templates

1.4.6 LANSA Built-In Functions (BIF)

1.4.7 "Compiling" RDML

1.4.8 RDML Benefits

#### Also See

**RDML Commands** in the *Technical Reference Guide* 

**RDMLX** Commands in the Technical Reference Guide

1. Getting Started with LANSA Development

# 1.4.1 Why a 4GL / RDML?

LANSA's RDML (Rapid Maintenance and Development Language) allows a developer to create a High-Level Application Definition or abstraction of your business application. This definition is independent of the operating platform. Defining logic with a 4GL makes your programs independent of the 3GL layer. A 4GL will eventually generate some type of 3GL code which is compiled and executed, but the developer need never see or modify the generated 3GL. In fact, it should not matter to the developer which 3GL is used. A High Level Application Definition should allow you to generate or regenerate the same applications in different languages to make your applications portable.

Why use a 4GL / RDML:

- High Productivity (Reduce Backlog): A single RDML command can replace hundreds of lines of 3GL code.
- Reduce Maintenance Effort: High-level constructs and less code make maintenance much easier.
- Platform Independent Definition: RDML can be generated into multiple languages to support multiple platforms.
- Portable & Durable: RDML is platform independent and can be extended, without impacting the developer, as 3GL languages improve and change.
- Focus on Business Application Logic: RDML is easy to use so that the developer can focus on the business logic instead of the programming language.

As your business and technology changes, LANSA's RDML protects your application programming efforts.

### ↑ 1.4 LANSA RDML

### 1.4.2 RDML Structures

This example of RDML code will display a data entry screen to allow Customer data to be written to a Customer file. It illustrates some of the key 4GL structures in LANSA's RDML and shows how these structures provide productivity and maintenance benefits:

FUNCTION OPTIONS(\*DIRECT) GROUP\_BY NAME(#CUSTOMER) FIELDS (#CUSTNO #NAME #ADDR1 #ADDR2 #CITY #STATE #PHONE) BEGIN\_LOOP REQUEST FIELDS(#CUSTOMER) DESIGN(\*DOWN) INSERT FIELDS(#CUSTOMER) TO\_FILE(CUSTMST) CHANGE FIELD(#CUSTOMER) TO(\*DEFAULT) END\_LOOP

A single FUNCTION statement controls the execution of your programs without complex external programs or parameter setting.

The GROUP\_BY statement defines an abstract construct, a field list which can be used in other RDML statements. The field #CUSTOMER represents a list of fields #CUSTNO #NAME #ADDR1 #ADDR2 #CITY #STATE #PHONE. Instead of typing each field name in subsequent statements to request, write or change data, the field #CUSTOMER can be used in its place. New fields can be added to the GROUP\_BY and will automatically be used wherever the #CUSTOMER field is used.

The REQUEST statement is all that is required for a complete screen to be created with all message and error handling. This single command replaces hundreds of lines of 3GL code. It controls the complete user interface and can build a 5250 screen for an IBM i or GUI screen for a PC running Windows. Screen characteristics are controlled by the repository and by a set of statement parameters such as DESIGN(\*DOWN) for how screen fields are designed.

The INSERT statement is used to write data to the Customer file. You will notice that no validation rules and error handling logic appear in the program. These are all handled by the repository and the parameters of the INSERT command.

The CHANGE statement combined with the GROUP\_BY statement replaces at least 7 lines of 3GL code which would be required to reset the individual fields. The use of \*DEFAULT means that the default field values are based on the

repository field default values when fields are reset. The developer does not have to hard code these values or read them from a file. There is less code to maintain. Changes can be made to the repository without changing this RDML code.

To add a new field such as #POSTCD to a Customer data entry screen, you simply add the field to the GROUP\_BY command and recompile. With LANSA's RDML, it is just one change to one line of code. It is that simple to code and maintain functions with RDML.

1.4 LANSA RDML

## **1.4.3 Programming Productivity**

Perhaps what is most important about the LANSA 1.4.2 RDML Structures is what does NOT appear. If you compare this LANSA program to programs written in a language such as RPG or C, you will notice that this program does not contain many lines of code required by other development tools and languages.

For instance, the LANSA program does not require explicit statements for:

- file declarations with open and close statements
- variable declarations
- validation rules & error handling routines
- function key & menu processing logic
- indicators and other screen processing controls
- and much more.

Because LANSA is a repository-based 4GL, many of these requirements are met by the 1.2.3 Repository and RDML combination. Files, variables, validation rules, etc. are all defined to the repository. The LANSA RDML commands automatically handle screen I/O and error processing. Again, a simple seven line RDML program is equivalent to a several hundred line program written in RPG or C.

1.4 LANSA RDML

# 1.4.4 Where is RDML Used?

RDML is used for many different purposes in the LANSA development environment. You can use RDML for the following:

### **Application Programs (LANSA functions) and Components**

This is the most common use of RDML. Developers use RDML to build the application logic within their business programs. Components created with Visual LANSA will also use RDMLX commands, which are specific to component development.

#### **Trigger Functions**

RDML can be used to define database trigger programs so that they can be ported with an application. If a trigger program is coded in a 3GL (such as RPG on the IBM i), it would have to be rewritten (recoded as C), when the application moves to a new database.

#### **Called Validation Functions**

Complex validation rules in the LANSA Repository can be coded using RDML.

#### **Built-In Functions**

RDML can be used to create specialized Built-In Functions to perform a specific repeated operation.

#### **Stored Procedures**

RDML can be used to define stored procedures so that they can be ported with an application. If a stored procedure is coded in a 3GL, it may have to be rewritten when the application moves to a new database.

#### **System Variables**

System variables are special variables which can be accessed by LANSA functions. For example, a date, time or company name can be a system variable. RDML can be used to derive the values of these variables.

RDML can execute on the client or server. It is a high-level, platform independent definition of your application logic. A platform independent development language is critical to enable deployment of a repository on more than one platform.

### ↑ 1.4 LANSA RDML

## **1.4.5 LANSA Application Templates**

LANSA's application templates are one of its most powerful features. An application template can be thought of as an RDML code generator. It uses a simple programming language. When executed, templates will automatically create standard blocks of RDML code based on a question and answer session. For example, a pop-up window or prompting function can be created using a template. You simply tell the template the particular file and fields that you want included in the window, and it generates the necessary RDML commands to build the pop-up window function/program.

Templates are extremely important as they ensure consistency in function design, operation and structure. Templates increase productivity and reduce coding errors. Customizing templates allows you to enforce site application development standards. Templates are an efficient means of generating commercial grade applications.

LANSA ships over 40 ready-to-use templates to help you with application development. Specific online help is available for each shipped template. Templates are written in a LANSA template language so new templates can be created by the developer, and existing templates can be modified to meet your site standards.

Where possible, it is recommended that you start with an application template when building your application programs. Building block templates can also be used within the function editor to create parts of a function.

#### Also See

Application Templates ↑ 1.4 LANSA RDML

# 1.4.6 LANSA Built-In Functions (BIF)

When you are developing complex computing applications, there is often a need to use some specialized set of programs or APIs (Application Programming Interfaces). For example, you may need to access fax, bar code or imaging technologies. LANSA provides a standardized interface for interacting with these technologies using LANSA Built-In Functions (BIFs).

Built-In Functions are a standardized and published set of specifications for calling external programs. By using a specific calling interface, management of the programs becomes very simple. The BIF interface allows developers to create a single set of code for multiple platforms. For example, a fax BIF can be called from an RDML program. On the IBM i, the BIF may interface with a FAX/400 program but on the PC the BIF may interface with a WIN/FAX program. The BIF allows a standard "plug-in" to be defined by the developer. LANSA also includes sets of standard BIFs to perform specific types of common operations. BIFs for date manipulation such as FINDDATE, and BIFs for string manipulation such as SCANSTRING, etc. are shipped as part of the LANSA development environment. LANSA also provides a set of BIFs such as CONNECT\_SERVER or CALL\_SERVER\_FUNCTION which allows you to quickly and easily build your client/server applications.

Other vendor or OV\_BIFS are also available. These BIFs are shareware which have been created by other LANSA developers. For example, a PC-based OV\_BIF called OV\_PASTE\_CLIPBOARD has been created for pasting information to a Windows clipboard.

BIFs can be created by any developer. They can be written for just one platform or they can be designed for use on multiple platforms. For example, you can create a BIF to interface with an existing RPG program on the IBM i, or you could create a BIF which interfaces with a C program used on more than one platform. BIFs can also be written using RDML.

#### Also See

Built-In Functions List in the *Technical Reference Guide* Create Your Own Built-In Functions in the *Application Design Guide*. ↑ 1.4 LANSA RDML

# 1.4.7 "Compiling" RDML

To execute a LANSA function, the RDML code is translated into an High-Level Language (HLL), which is then compiled into an executable program object. RDML is not an interpretive language. It creates a program object which is native to the execution environment so your application has optimum performance characteristics for that platform.

The RDML compile process and its generation of an intermediate HLL code is completely transparent. You do not need to see the generated High-Level Language code and should never modify this code. LANSA is not a code generator. It is a complete application development suite using a repository with a 4GL. Modification of the HLL code would destroy the integrity of your highlevel application definitions in LANSA. Once you modify HLL code on a platform, you restrict the code to execution on that specific platform only.

LANSA uses different High-Level Languages depending on the targeted execution platform. For example, RPG and C/C++ provides the fastest execution for the IBM i platform. For Windows and Linux, ANSI C/C++ is used. For the Web, you might use C/C++, XHTML, XML and CGI/Apache Plug-in/IIS Plug-in. **LANSA's ability to use multiple languages is key to creating a platform independent application solution**.

LANSA uses platform dependant compilers to produce executable program objects and has found that these compilers differ in their ability to handle large amounts of HLL code. Under certain conditions, the compiler can hit internal limits and fail to produce an executable program. To help understand this situation, the following points should be taken into consideration:

- To minimize the possibility of hitting a compiler's internal limits, LANSA restricts the number of lines of RDML/RDMLX code must be translated as a unit. RDML Functions are limited to 5000 lines while Components and Full RDMLX Functions are limited to 32000 lines.
- LANSA commands generate different amounts of HLL code. Workstation commands (like DISPLAY, REQUEST and POP\_UP) and I/O commands (like FETCH, SELECT and UPDATE) generate much more HLL code than commands like CHANGE, IF and BEGIN\_LOOP.
- The compiler being used. For example, a Full RDMLX LANSA Component consisting of 18000 lines of RDML was translated into a 28MB C/C++ HLL file and this file was successfully compiled by the Microsoft 32-bit compiler, but the component had to be reduced to 9000 lines in order to be compiled

by IBM i's compiler.

- Enabling RDML/RDMLX debugging. When 4GL debugging is requested, additional HLL code is produced which can cause the compiler limits to be exceeded. For example, using the HLL code produced by the 9000 lines of RDML from the previous example, IBM's IBM i C/C++ compiler failed when 4GL debugging was requested, but was successful without it.
- Enabling observability (3GL debugging). Sometimes it is necessary to debug an executable program using the HLL code produced by LANSA. Requesting this feature can also impact the amount of HLL code that can be compiled. For example, when observability was selected, 9000 lines of RDML had to be reduced to 4500 lines before IBM i's compiler would compile successfully.

1.4 LANSA RDML

## 1.4.8 RDML Benefits

LANSA's RDML provides the following benefits:

- RDML provides powerful language constructs which dramatically increase productivity and reduce application maintenance.
- LANSA's application templates provide high productivity as they generate standard blocks of error free code which can be customized to meet your site standards.
- RDML requires fewer lines of source code for easier application maintenance. One line of RDML can be equivalent to over 100 lines of 3GL code.
- With RDML, one application definition can support multiple platforms as RDML can generate multiple languages. RDML makes applications portable and durable.
- LANSA's Built-In Functions provide a simple interface with external programs.
- The same RDML language can be used on client and server (and for client/server). This provides greater flexibility for your application deployment and execution.
- One language means one set of skills. You can develop for multiple platforms without having to learn multiple tools.
- RDML protects your information technology investment! RDML provides a high-level application definition for your business application. You can focus on the business logic and not the programming language. As programming languages evolve, RDML will evolve to keep your technology current.

LANSA's RDML is a key part of the LANSA architecture and technology solution.

1.4 LANSA RDML

# 1.4.9 RDMLX Benefits

RDMLX is an extension of LANSA's RDML. RDMLX development is done using Visual LANSA. In addition to supporting all LANSA RDML features, RDMLX enabled partitions support RDMLX language features hich include:

- use of intrinsic field methods
- use of function libraries
- use of expressions in many parameters and properties
- enhanced RDMLX command support such as assignment statements
- simplification of RDML statements due to the removal of quotes.

For example, an intrinsic field method can be used instead of using a Built-In Function to convert a number to a string. A numeric field such as #SALARY can be referenced as #SALARY.asString as in this example:

#PHBN\_1.Left := #PHBN\_2.Right + 10

```
#FullAddress := RightTrim(#Address1) + ' '
+ RightTrim(#Address2) + ' ' + RightTrim(#Address3)
```

SET #Com\_Owner Left(#Com\_Owner.Left + 2) Top(#Com\_Owner.Top + 2)

IF ((#QUANTITY \*LT 10) \*AND (#MEASURE \*GT 10.5))

IF (#PHBN\_2.Caption.Substring( 1, 1 ) = 'A')

```
#STD_NUM := Length(Trim(#Com_Owner.StringMethod(#Address1 #Addres
```

DOUNTIL (#I >= 11)

These language features may only be used in RDMLX Objects.

In order for an existing component to use these language features, the partition and components must be *Enabled for Full RDMLX*.

#### Also see

**RDMLX** Partition Settings in the Visual LANSA Administrator's Guide.

### **1.5 Developer Resources**

The following is a list of valuable resources for LANSA developers:

- 1.5.1 SET (Samples Examples Templates)
- 1.5.2 Online Tutorials
- **1.5.3 Personnel Demonstration Application**
- 1.5.4 Training and Consulting Services
- 1.5.5 LANSA Web Site and Support
- 1. Getting Started with LANSA Development

## **1.5.1 SET (Samples Examples Templates)**

The SET materials are composed of hundreds of examples and templates designed to help you create LANSA-based solutions rapidly and effectively. SET is intended for both LANSA application designers and developers. It is designed for people working with any LANSA product.

SET covers many application areas from simple 'How do I?' examples to complete business application scenarios. It is installable as ready-to-run applications, including all source code.

Following are some of the Visual LANSA examples available:

- Some Ideas
- Communication and Connectivity
- Windows Server Systems
- Multi-Tiered Applications
- Miscellaneous Examples
- MS-Excel Integration Examples
- MS-Word Integration Examples
- Web Browser Integration Examples
- Visual Basic Integration Examples
- Lotus Notes / Domino Integration Examples
- Email Integration Examples
- Using the SPACE Built-In Functions
- Base Forms and Reusable Components.

It is strongly recommended that you review the SET materials.

SET materials can be accessed using the *SET* button on the toolbar, or from the main documentation *Directory* page.

1.5 Developer Resources

## **1.5.2 Online Tutorials**

Online Tutorials are supplied with the LANSA products as part of the core documentation. These tutorials provide step-by-step instructions to train new LANSA developers on how to use LANSA products and features. Online tutorials provide an excellent introduction to LANSA and should be the starting point for developers as they learn new products. For example, to learn how to build Internet applications with Web Application Modules, refer to the WAM Tutorials in the *Web Applications Module (WAM) Guide*.

LANSA Online Tutorials can be accessed using the *Tutorial* button on the toolbar, or from the main documentation *Directory* page.

If you are planning to use the Online Tutorials, you must install the 1.5.3 Personnel Demonstration Application.

#### Also See

- 1.5.4 Training and Consulting Services
- 1.5 Developer Resources

## **1.5.3 Personnel Demonstration Application**

LANSA ships a very simple Personnel System application database that includes demonstration applications using functions, forms and web-based solutions. This demonstration system is used as the basis of all online training. To install the Personnel Demonstration Application, refer to Partition Initialization in the Administrator Guide.

Following is a brief description of the Personnel System application:

A business has a very simple Personnel System. The Personnel System allows the company to identify the employees in the company based on the part of the company where the employee works. The Personnel System lists details about the employees and details about their specific skills.

The company has a simple organizational structure. It is divided into departments such as Administration, Audit, Information Services, Legal, Travel, etc. Each of these departments may have one or more sections such as Accounting, Purchasing, Sales, etc. The Department table (DEPTAB) stores the list of departments. The Section table (SECTAB) is used to store the sections within each department.

The Personnel Master file (PSLMST) stores details about each employee. For example, the employee's name, address, and telephone number are stored in this master file. As each employee works in a section of a department, this information is also stored in the Personnel Master file.

Each employee also has a list of skills. For example, an employee might have Cobol, C and C++ programming skills or management and administration skills. A Skills table (SKLTAB) is used to store the skill codes. A Personnel Skills file (PSLSKL) stores the specific skills of each employee.

The Personnel System is a very simple system. It has 5 files as described above. The physical database layout is described in Physical Database Map of Personnel System.

**Historical Note:** This system was created in 1987 as one of the very first LANSA demonstration and training systems. The LANSA repository and RDML functions created for this original system have been used on a System 38, AS/400, iSeries, System i, IBM i, Windows, Linux and other platforms. This original system has been left virtually unchanged to show how LANSA has been able to protect your investment in your application systems.

### ↑ 1.5 Developer Resources

# **1.5.4 Training and Consulting Services**

# LANSA Training

LANSA offers a variety of training choices to suit your needs.

**1.5.2** Online Tutorials are supplied with the LANSA products as part of the core documentation. For example, to learn how to build Internet applications with Web Application Modules, refer to the WAM Tutorials in the *Web Application Module (WAM) Guide*. Tutorials can be accessed using the Tutorial button on the toolbar, or from the main documentation Directory page.

Your LANSA distributor offers classroom education where you will learn from the experts. LANSA courses are developed and taught by experienced professionals who have years of application development experience plus an intimate knowledge of the LANSA technology. They bring their real world experience into the classroom for your benefit. LANSA courses are offered publicly at multiple locations. Courses can also be conducted at your company's facility or a location of your choice. Refer to the LANSA Web site at www.lansa.com for information about LANSA courses.

# **Consulting Services**

Your LANSA distributor offers a Consulting Services Division that will assist you to build and adapt mission-critical applications using the LANSA family of products and tools. While LANSA products offer immediate productivity gains, there are often circumstances where business can benefit by supplementing the LANSA products with professional services.

### Also See

Customer Service in the Administrator Guide

1.5 Developer Resources

### **1.5.5 LANSA Web Site and Support**

LANSA provides a great deal of product information on its World Wide Web site at www.lansa.com. The latest product information, customer stories, events, and developer tips and techniques are posted regularly. Information exchange groups are also available so that you can receive email on specific forum topics.

To contact your local LANSA support group and to identify the information that may be requested to investigate your LANSA issue, refer to the LANSA Help Desk (www.lansa.com.au/support/helpdesk.htm) on the LANSA Web site.

#### Also See

Troubleshooting in the Administrator Guide.

1.5 Developer Resources

# 2. Creating LANSA Applications

Before creating a new LANSA application, review the following topics:

- 2.1 Types of LANSA Applications
- 2.2 Application Design
- **2.3 Application Development**
- 2.4 Multilingual Applications

#### Also See

Getting Started with LANSA Development

# 2.1 Types of LANSA Applications

LANSA can be used to create a wide variety of applications. How you will use LANSA will depend upon the type of application being created and the model which best suits your business requirements. Review the following topics:

- 2.1.1 LANSA Application Characteristics
- 2.1.2 LANSA Application Models
- 2.1.3 Universal Model
- 2.1.4 Windows Optimized Model
- 2.1.5 Web Optimized Model
- 2.1.6 Open Model
- 2.1.7 Types of Business Applications
- 2.1.8 Mixing Models
- 2.1.9 Query & Reporting with LANSA

### Also See

Developer Roadmap

 $\ensuremath{\Uparrow}$  2. Creating LANSA Applications

# 2.1.1 LANSA Application Characteristics

LANSA is best described as follows:

• LANSA is an integrated family of highly productive design, development and maintenance products for multi-platform host and network applications.

Described another way:

• LANSA allows you to design and build a business application which you can execute on multiple platforms in a host or network mode.

The key words to remember in the LANSA definition are:

- **integrated family** LANSA is a family of products which have been built to work together. LANSA allows you to share and reuse application information.
- multi-platform

LANSA supports the IBM i, Windows, and other platforms. When you develop with LANSA, you are platform independent.

• network

LANSA supports many different modes of processing for a network framework including client/server and Web-based distributed processing.

Understanding the importance of these characteristics will help you to understand the LANSA Architecture, structure of the LANSA product family, and the types of application that can be created.

 $\ensuremath{\Uparrow}$  2.1 Types of LANSA Applications

# 2.1.2 LANSA Application Models

Once you have created an application with LANSA, the application can execute on a wide range of host platforms (IBM i, Windows, Linux, etc.). A LANSA application can execute as a Web-based application or as a Windows-based client/server application. These deployment options are all possible using a single set of LANSA application definitions. You might also choose to build specific applications with LANSA which are optimized for Windows or optimized for the Web.

Using the various products in the LANSA family, you can develop three styles or models of applications:

#### 2.1.3 Universal Model

The Universal application is characterized by the fact that the same application can execute on multiple platforms (IBM i, Windows, etc.) and on the Web by simply recompiling the application definition.

#### 2.1.4 Windows Optimized Model

Windows Optimized applications are highly graphical point -and-click type applications which will only execute in environments (such as Windows) which support the event-driven processing model. This type of application uses a repository-based component technology built with Visual LANSA.

### 2.1.5 Web Optimized Model

Web Optimized applications are created specifically for the Internet deployment environment. This type of application may use specialized HTML, JavaScript, XML or XSL. They are designed for Internet devices and must cater for the "browser paradigm" which includes features like a Back key.

 $\$  2.1 Types of LANSA Applications

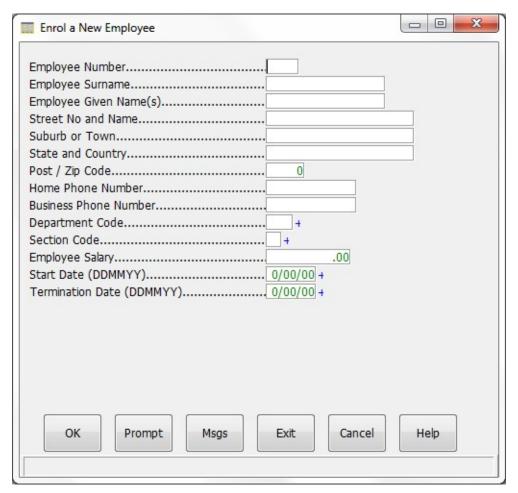
### 2.1.3 Universal Model

You may choose to develop a large portion of your LANSA applications using the Universal model. The application interface is described as Universal because it can be used in all LANSA products. For instance, a Universal application can be executed natively on the IBM i, under Microsoft Windows, over the Internet in a browser environment, or on any number of computing devices using XHTML or XML. The Universal model takes full advantage of LANSA's repository-based 4GL architecture. It is an excellent example of portability and platform independence.

Using the universal model, a LANSA application on an IBM i can appear like this:

Enrol a New Employee
Employee Surname
Employee Given Name(s)
Street No and Name
Suburb or Town
State and Country
Post / Zip Code 0
Home Phone Number
Employee Number
Business Phone Number
Department Code
Section Code
Employee Salary00
Start Date (DDMMYY) <u>0/00/00</u> ±
Termination Date (DDMMYY) 0/00/00 +
Fl=Help F3=Exit F4= Prompt Fl2=Cancel Fl4=Msgs

The same LANSA application, compiled for Windows or the Web, can appear like this:



However, because the user interface is Universal, it cannot be optimized to take maximum advantage of the unique characteristics of any single environment. The universal interface has some graphical capabilities (based on the IBM Windows Icon Mouse Pointer constructs). It supports features such as action bars, push buttons, radio buttons, drop downs, lists, etc. The programs are procedural and use a character-based application style. Under Windows, a Universal application does not have the full range of event-driven capabilities. When you develop a Universal style application, you can use either the LANSA for i or the Visual LANSA development environments. The applications use a full range of repository features and RDML commands. The Universal application is based on LANSA RDML Functions. (Refer to Creating Application Using Functions.) Universal applications do not require Full RDMLX features. (Refer to RDML and RDMLX Partition Concepts in the

Administrator Guide.)

To deploy to specific platforms, you simply use the Visual LANSA development environment to recompile and deploy the application. Developing applications under the Universal model provides the greatest range of flexibility for your application.

Universal applications share a common repository with Windows Optimized and Web Optimized applications.

 $\ensuremath{\Uparrow}$  2.1 Types of LANSA Applications

# 2.1.4 Windows Optimized Model

A Windows Optimized LANSA application is created to make full use of the features of high function graphical user interface environments. Windows Optimized application components are fundamentally different from Universal applications because they must be event driven to support high function graphical user interface controls such as grids, tree views, graphs, charts, drag and drop, multimedia, etc.

Object Edit View Tools			
Administration	- 渣 🖬 🗙 🛛 👗 🖆	3 🖆	
All Objects	Employee Information Ve	eronica Brown	
Vehicle Maintena Internal Auditing	Employee Details Ski	lls Email	
Fleet Administration Accounting	Employee Number	A0070	A COMPANY
Administration	Employee Surname	Brown	e Arrenta
Purchasing Group Accounts	Employee Given Name(s)	Veronica	68
Accounting	Business Phone Number	(02) 9647 2788	A STOL
Administration Financial Control	Home Phone Number	(02) 9609 4627	JEAN A
Purchasing	Start Date (DDMMYY)	28/01/90	
Information Services Accounting	Termination Date (DDMMYY)	0/00/00	
Adminsitration	Street No and Name	12 Railway Street	
<ul> <li>Development</li> <li>Purchasing</li> </ul>	Suburb or Town	Baulkham Hills	Enlarge Photo
🗅 🛄 Legal	State and Country	NSW	Save
Mgmt. Information Marketing	Post / Zip Code	2153	Cancel
Sales & Distribution Travel	Employee Salary	50,125.00	Delete

Windows Optimized applications executed on Microsoft Windows workstations or under environments which support the event-driven processing model. These applications can be written to use a variety of servers in a client/server environment.

Windows Optimized applications are created using the Visual LANSA development environment. A Windows Optimized application uses componentbased technology. The components are stored in the LANSA Repository. The components use the same RDML skills, but also have specialized componentbased programming. (Refer to Creating Application using Forms.)

Because Windows Optimized applications are component-based and support the

event-driven paradigm, the applications are designed in a very different manner from procedural applications. Components support properties, events and methods and follow an object-oriented paradigm.

Windows Optimized applications share a common repository with Universal and Web Optimized applications.

 $\ensuremath{\Uparrow}$  2.1 Types of LANSA Applications

# 2.1.5 Web Optimized Model

A Web Optimized application is developed to cater for the unique needs of the Internet, HTTP and the HTML browser. For example, the Hyper Text Transport Protocol (HTTP) is a connectionless protocol, so there is no "transaction state" supported. Browsers have a Back key which allows users to process screens out of sequence.

Using LANSA, you can code your applications specifically for the Web execution environment by using special RDML options. These options mean that the Web Optimized applications can only be executed via a Web browser. You can develop Web Functions that allow you to customize the LANSA generated XHTML and JavaScript for your Web application. You can also develop component-based Web Application Modules (WAMs) using XML/XSL that support a variety of technology services.



Web Function applications can be created using the LANSA for i or the Visual LANSA development environments. (A Windows-based HTML editing tool is provided by LANSA.) Web Application Modules can be created using Visual LANSA.

Web Optimized applications share a common repository with Universal and Windows Optimized applications. Web Functions applications may require specialized HTML or JavaScript coding skills. While WAMs use similar RDML skills, but also have specialized component-based Web programming skills.

LANSA for the Web provides a flexible deployment architecture. You can use a variety of Web Servers (IBM i, Windows, Linux) in combination with Windows or IBM i Data/Application Servers. You can use single tier (Web Server and Data/Application Server on the same machine) to multi-tier deployment models (Web Server and Data/Application Server on different machines).

#### Also See

Web Development 1 2.1 Types of LANSA Applications

# 2.1.6 Open Model

Once you have developed an application with LANSA, you may wish to develop extensions to your LANSA-based applications using other PC-based development tools, or you may wish to integrate the data in your LANSA-based application with other applications in your organization. The Open Model supports this type of development and integration.

LANSA Open allows you to choose your PC-based development tool while safely accessing data on the server using LANSA's Repository. The LANSA Open middleware provides fast access to the LANSA Repository where your business rules and logic are consistently enforced. You can call LANSA Open from any tool that can call a DLL. For example, Visual Basic or Delphi developers could access data in your LANSA applications.

Using LANSA Integrator's Java services frameworks, you can easily integrate data between any applications using different formats on different servers. You can use XML or a variety of data formats for data exchange. LANSA Integrator's Java Service Manager supports B2B technologies such as SOAP, IBM MQ Series, XML Parsing, XSL Transformation, FTP, HTTP, HTTPS, SMTP, and POP3. End-user data transfer tools such as the User Agent (which uses a HTTP protocol) enable data transfer using standard PC-based file formats while Java development tools such as Remote Function Invocation (RFI) simplify data transfer to the server.

The Open Model provide opportunities for inter-operability of your LANSA applications with other applications within your organization.

#### Also See

Open Development LANSA Integrator Guide LANSA Open Guide 1 2.1 Types of LANSA Applications

## 2.1.7 Types of Business Applications

There has been an evolution of office roles as technology has enabled companies to extend their core business system out to business partners and customers. For example, business partners may be able to check stock level using client/server applications executing from their own offices. Customers may be able to order products on the Internet. Applications are no longer used solely by the employees of a company.

Following are some general categories of office roles:

#### **Process-driven roles**

Process-driven roles often involve a rapid repetition of a single task such as dedicated data entry. For example, a trained employee may be entering orders which were faxed by customers. LANSA's 2.1.3 Universal Model is a good match for process-driven roles.

#### **Customer-driven roles**

Customer-driven roles involve a varied office role. There may be frequent telephone interruptions. There is frequent task switching as the user may need to write letters or analyze graphs and reports. The integration of other office tools becomes important. LANSA's 2.1.4 Windows Optimized Model is a good match for customer-driven roles.

#### **Direct customer Internet access**

Customers may be directly accessing your applications using the Internet. This may involve very occasional use by a customer who is untrained in your application. The customer may be frequently switching between tasks and may not complete a transaction. In this situation, a 2.1.5 Web Optimized Model is the best solution.

#### Integration: B2B and A2A

Finally, your applications may evolve into integrated Business-to-Business (B2B) and Application-to-Application (A2A) solutions. In this situation, the 2.1.6 Open Model and LANSA Integrator is used to support business transactions.

### Also See

- 2.1.8 Mixing Models
- ↑ 2.1 Types of LANSA Applications

# 2.1.8 Mixing Models

Different users and different applications will have different needs. Within your business applications, you may need to have a series of diverse and sometimes functionally overlapping user interface and application styles for different parts of your core business applications.

Typically, about 70% of an application's total user interface requirements can be handled purely in the 2.1.3 Universal Model. Approximately 15 - 20% will significantly benefit from being 2.1.4 Windows Optimized Model and 5 - 10% will benefit from being 2.1.5 Web Optimized Model or may use the 2.1.6 Open Model.

Of course the mix will vary somewhat with the specific type of business application. Some applications might have 90% Web Optimized and 10% Universal. Other applications may require 100% Windows Optimized.

It is important to remember that the repository-based 4GL architecture is the key to the LANSA solution for your business applications. LANSA embeds the largely hidden core of the application (its business rules, event triggers, remote procedures, etc.) in the repository which supports a wide range of differing user interface requirements. Your investment in the repository is your investment in the future.

By using a 4GL to develop your applications, you have a much greater range of interface choices as demonstrated by the Universal application style. As new languages and platforms are introduced and evolve, the importance of the LANSA Repository and RDML grows.

In summary, it is your choice. You can use a number of LANSA products to build a variety of applications to meet your business needs.

LANSA provides an integrated family of products which share a common repository-based 4GL architecture. A LANSA developed application offers a company a wide range of user interface style choices to meet the demands of the application.

LANSA for i builds applications for the Universal user interface style. It is ideal for your batch, reporting and dedicated data entry type applications. The application can be deployed to the Web or to Windows by simply recompiling.

Visual LANSA builds applications for both the Universal, Windows Optimized and Web Optimized user interface styles. You build Windows Optimized applications by using Visual LANSA component technology. LANSA for the Web allows you to deploy your existing Universal applications on the Internet or using an Intranet. You can also create Web Optimized applications using Web Functions or WAMs for specialized Internet/Intranet applications.

LANSA Open allows you to choose a PC-based development tool but still share in the benefits of the repository architecture.

LANSA Integrator allows you to exchange data with other applications on other servers to build application-to-application or business-to-business type applications.

It is your choice.

#### Also See

↑ 2.1 Types of LANSA Applications

# 2.1.9 Query & Reporting with LANSA

Finally, once you have completed your application, LANSA offers an end-user productivity tool for query, reporting and charting of LANSA application data. It also allows you to import the data to spreadsheets or other PC tools for analysis. It makes use of the LANSA Repository technology, thereby protecting your data, while providing fast and efficient access.

With LANSA, you can centrally define and control query access on the server which helps you to avoid time-consuming and error-prone LAN replication of files and definitions. Fields are given business names that users find easy to understand. Also, navigation rules between files are defined, making it easy for users to find the data they need.

Designed for end-users, this tool is easy-to-use and can help to significantly reduce your application backlog by allowing users to create their own specialized reports. Users can easily create sophisticated reports and charts. For example, complex queries can be created by simply dragging and dropping fields from files. Retrieved data can be plotted and charted using many different formats. A point-and-click application generation facility lets you package multiple promptable queries and reports into a single icon. It is the ultimate in simplicity for any end user.

#### Also See

Client Development LANSA Client Guide 1 2.1 Types of LANSA Applications

# 2.2 Application Design

The focus of this guide is application development. The following topics are provided as a simple overview of application design concepts:

2.2.1 Structured Methodologies

2.2.2 Using a Modeling Tool

2.2.3 Importance of Design Phase

#### Also See

LANSA Application Design Guide

1 2. Creating LANSA Applications

## 2.2.1 Structured Methodologies

LANSA is methodology independent. You can select from a variety of structured methodologies and use the methodology of your choice with LANSA. LANSA does not force a developer to use one specific structured methodology. For example, you might use:

#### **Software Engineering Methodologies**

Software Engineering is a "process driven" design approach which focuses on the flow of information within an organization and the transactions which occur during the flow. Emphasis is placed on business activities and procedures. Process driven design looks first at what procedures or operational tasks a business performs and then looks at the data that is used and how it is structured.

#### **Information Engineering Methodologies**

An Information Engineering design approach is based on the concept that a business is built from the structure of its data. The computer stores and structures the information and provides programs to manipulate the data. Business requirements are analyzed in terms of information needs. Emphasis is placed on a corporate wide data model. Once this is determined, the processes which manipulate the data are reviewed.

#### **Object Oriented Methodologies**

The Object Oriented Design approach is based on the concept that a user's activities center on the manipulation of objects used by the business. Object oriented designs emphasize the objects that a user works with to complete a business task. The approach does not place first priority on the structure of the data or the procedures which the business performs. All objects that a user requires to perform a business task are identified. The data related to the object and the actions performed using the object are defined. Both data and procedures are modeled as they relate to each other and to the objects.

Many business simply use a traditional Waterfall Methodology with LANSA. LANSA provides a very flexible development environment that can accommodate a variety of analysis, design, development, testing and deployment strategies.

### 1 2.2 Application Design

# 2.2.2 Using a Modeling Tool

Logical models benefit the System Analysts, Designers, Developers and End-Users as well as the final application. For example, using a logical model will:

- highlight problems earlier so less rebuilding is required
- make the data in the application easier to understand by providing the "big picture" of the information
- document relationships within the application for the Builder
- re-use the work done by the system designer so that the developer does not have to repeat it
- improve productivity by creating fields, files, validations, access routes for using templates, help text, etc.

LANSA provides a tool for modeling. Refer to:

• Visual LANSA Logical Modeler Guide

Although you do not have to use a logical data model or object action model to work with LANSA, there are significant advantages to modeling. We strongly recommend using the LANSA modeling tool and modeling techniques to develop new applications.

1 2.2 Application Design

# 2.2.3 Importance of Design Phase

The design phase focuses on developing a physical plan which can be used to construct the application system. The inputs to the design phase may include some type of systems and/or user requirement document from the analysis phase. The outputs from the design phase will include a database definition and a series of functional specifications which define the framework of the application.

Most approaches to designing a system begin with the data and then look at the procedures or processes which manipulate the data. You must know the details of what to manipulate before you specify how to manipulate it.

Design work should begin with a "logical" or "conceptual" model rather than a "physical" design. Logical models look at data and procedures in abstract terms, e.g.. there is a customer which places an order for product. The physical design looks at implementation, e.g.. the OP999X order program will use the CUSTMST file and PRDTMST file.

In many instances, applications are created relying solely on physical designs. The problem of poor or lacking design can be made even worse with the introduction of productivity tools such as CASE. The development environments become so productive that people skip the design phase and just build a "solution".

The following are some questions which will help to identify applications which suffer from lack of conceptual design:

- When you maintain a program, do you refer to a design document, or do you have to review the code to understand what the program is doing?
- Do you have any documentation for your system? And is there anything to document other than the code? Are your "quick fixes" becoming common code?
- Do screen designs, program names, field names, etc.. change radically from one application to another? Is it easier to rewrite a program than to change it?
- Do the end-users ever see the application before it is put into production?

The advantage of using a productivity tool is that the improved productivity should provide more time for developers to design before they build, or THINK before they ACT.

#### Also See

LANSA Application Design Guide↑ 2.2 Application Design

## 2.3 Application Development

This section of the guide provides an overview of LANSA application development.

- 2.3.1 Types of Development
- 2.3.2 Start with Standards
- 2.3.3 Using Existing Applications
- 2.3.4 Building LANSA Applications
- 2.3.5 Step 1. Building the Repository
- 2.3.6 Step 2. Building the RDML

#### Also See

Developer Roadmap

1 2. Creating LANSA Applications

# 2.3.1 Types of Development

When you begin to develop applications with LANSA, your projects may involve one or more of the following categories:

#### **Enhancing Packaged/Existing Applications**

Most businesses have a packaged application or some type of existing legacy application. You can use LANSA to extend these packages and applications. You can load your existing databases into the LANSA Repository and create new types of applications.

#### **Re-engineering Business Applications**

At some point, your business may decide to re-engineer an older legacy application to enhance business functionality and take advantage of the latest technology. LANSA provides some rapid prototyping tools to help accelerate the re-engineering process. Because LANSA applications share a common repository, re-engineering into new technologies can be done in a fraction of the time.

#### **Custom Development of New Applications**

As businesses change, there are needs for new application systems. By properly structuring your new applications with LANSA, you can maximize your business systems investment.

LANSA can be effectively used in each of these situations. However, the techniques or way you develop with LANSA will be slightly different in each situation.

Also when creating your applications, you must consider the type of application and its required user interface. Will the application be used from a host system, such as the IBM i, for dedicated data entry? Will the application be integrated with other desktop applications? Will the application be used directly by your customer over the Internet? Again, LANSA allows you to build four categories of applications - Universal, Windows Optimized, Web Optimized and Open. (Refer to 2.1 Types of LANSA Applications.)

You must understand your application requirements so that you select the best LANSA solution to fit your application. (Refer to 2.1.7 Types of Business Applications.)

1 2.3 Application Development

## 2.3.2 Start with Standards

When you begin to build your repository, you may use LANSA's modeling tools or you may manually define objects in the repository. In either case, you should start by creating a set of naming standards for objects (fields, components, files, access routes, etc.) used in the repository. You should also define a set of procedures for repository maintenance.

Standards are needed to protect your investment in the repository.

Rules or standards for driving an automobile were created so that all drivers know how other drivers will behave. For example, they expect cars to be on a particular side of the road and they know who should give way at an intersection. When the rules or standards are not followed, disaster can result and most often the damage is done to the one who did not break the rules.

The same is true of standards necessary for working with a repository. Creating and following standards will:

- protect the information in the repository
- control changes to your existing functions
- improve productivity of the developers
- reduce the maintenance effort for your applications.

The success of a project will be determined by the commitment of all staff to following the standards created at your site. We will review some object naming standards as new objects are created in the repository.

For details about controlling your development environment, refer to **Development Environment Administration** in the *Administrator Guide*.

#### Also See

LANSA Application Design Guide

1 2.3 Application Development

# **2.3.3 Using Existing Applications**

The LANSA Repository is not just for new application development. LANSA easily integrates with your existing applications. The LANSA Repository can be populated with existing application database files by utilizing a file load utility in the LANSA development environment.

LANSA has the ability to load existing IBM i or Windows file definitions into the repository. LANSA reads the physical file or table definition information to create the necessary field, file, and even logical file definitions in the repository. When an existing file is loaded, it is described as an "OTHER" file as opposed to a LANSA defined file.

LANSA does not recreate the physical or logical files. It simply defines their structure within the repository so LANSA will be able to access the same file used by your existing application. However, once defined in the repository, LANSA can now add all the benefits of the repository architecture to this older database.

For example, a Customer Master file from an Order Entry application can be loaded into the repository. New repository fields for the Customer ID Number, Customer Name, and other fields are created. The file's structure and logical files are identified. Once in the repository, the field definition information can be extended so that help text, prompts, consistent labels and descriptions are defined. Default values and display attributes can be specified. Business validation rules which were often stored in many different programs can now be entered and maintained in one central location. Relationships to other files can be defined so that the database schema is known to the development environment and end user tools. There is no change to the existing Customer Master file. The file is still actively being used as part of the Order Entry application, but it now has an extended definition in LANSA which is much more powerful.

The ability to use the LANSA Repository with existing databases allows you to extend your existing investment while building a more robust application for the future.

↑ 2.3 Application Development

# 2.3.4 Building LANSA Applications

A development environment using an Active Repository architecture improves the way developers create business applications. A central repository removes the redundant information from individual programs. The common business application information needs to be defined only once. With LANSA's integrated development environment, building an application is a simple two step process.

### **Step 1: Build the Repository**

The first step in building your application is defining the information about fields and files to the LANSA Repository. Again, you can load existing databases into the LANSA Repository. You have a choice of how many or how few of the repository features you use. The more business information you centralize in the repository, the higher the productivity and return on your development investment.

One of the key benefits of the repository is that it can be changed even after application programs have been created. You can continually add new rules and definitions into the repository. It does not have to have to be a "fully completed" repository before you can start program development.

### **Step 2: Build Your Application Programs**

Once the basic repository information has been entered, you can begin developing your application programs. These programs may be created using LANSA RDML or could be written using another language on another platform. As you begin programming, you may find additional logic that you wish to add to the repository using triggers or stored procedures. You can also use a cyclic process of refining repository contents as you first develop your programs. Many of the productivity and maintenance benefits are the result of how LANSA RDML uses the LANSA Repository.

The LANSA Repository and RDML work together to create a powerful integrated development environment. When developing your applications with LANSA, you use a two step process of developing your repository and then your RDML application logic. The development process is cyclical. You will often go back and enhance the repository as your business changes.

No matter what type of development or what type of application you are creating, you will need to develop the repository first. The repository is common to all products in the LANSA family. Your investment in the repository can be shared and reused. It is important to spend time to properly define the repository. It will form the core of your application for years to come.

Remember, LANSA's RDML is much more productive because it has been designed to reuse the information it knows is already defined in the repository. Before you begin to develop your RDML applications, you will need to make some decisions about the application style. The application style can impact the RDML you develop. Whether Universal, Windows Optimized, Web Optimized or Open, you can use your RDML skills to build the application. (Refer to 2.1.2 LANSA Application Models and 2.1.7 Types of Business Applications.)

1 2.3 Application Development

## 2.3.5 Step 1. Building the Repository

The first step in creating an application is building the LANSA Repository definitions. Before you begin to create any RDML functions (programs), you must have a properly defined repository. Remember, the repository centrally stores much of the application definitions so that it does not have to be coded into your application programs. The repository stores the information about the fields and the files that are used by the application. It stores details about the related business rules for these fields and files.

Briefly, there are three methods which you can use to build your LANSA Repository.

#### Manually Create the Repository:

If you are creating a new application (or even enhancing an existing application as described below), you can manually define the information to the repository. Remember to spend time to properly define the information in the repository. For example, when you create a field in the repository you are also designing screens and reports, setting up business rules, including help text, etc.

### Load Packaged Applications/Existing Database Files:

LANSA easily integrates with your existing applications. The LANSA Repository can be populated with existing application database files by using a file load utility in the LANSA development environment. Once defined in the repository, you can now manually add all the benefits of the repository architecture to this pre-existing database. (Refer to 2.3.3 Using Existing Applications.)

#### **Use A Modeling Tool:**

One of the fastest ways to build the LANSA Repository is 2.2.2 Using a Modeling Tool. Modeling tools can build all of your application database and related fields.

#### Also See

Controlling Repository Maintenance Steps for Developing with a Repository 2.3.6 Step 2. Building the RDML ↑ 2.3 Application Development

### **Controlling Repository Maintenance**

The LANSA Repository is a core component of the application development environment. A single change to the repository may significantly impact a wide range of objects in LANSA. Remember, a new field validation rule affects every function which accesses that field. If a field description is changed, it potentially impacts every screen that uses that field.

Maintenance of the repository can be simplified by setting up proper procedures in advance:

- Define and publish site standards so that all developers understand how the maintenance of the repository will be done.
- Enforce development procedures by using LANSA security to limit access to specific development options such as creating fields or files.
- Control and limit the access to the repository definitions so that changes are made in a controlled fashion and are being evaluated before they are made.
- Use the LANSA impact analysis tool to review how a change will affect your application.

When developing with LANSA for i, together with Visual LANSA in a client/server distributed development environment, you may have multiple copies of the repository. In this case, it becomes even more important that proper procedures and standards are enforced.

For more details, refer to the LANSA Application Design Guide.

#### Also See

Steps for Developing with a Repository ↑ 2.3.5 Step 1. Building the Repository

### Steps for Developing with a Repository

Following is an example of the steps which you might follow when developing an application using a Customer file.

- 1. Define Fields. The first step is to define the Customer fields to the repository. The field definitions can include descriptions, labels, defaults, help text, prompts, etc. This information will be used by the file definitions, RDML functions, etc.
- 2. Define/Create Files. Once all of the Customer fields are defined, they can be used to create a file definition. The physical file and any logical files can be specified, along with file triggers, attributes, etc. Once defined, the actual file/table can be created in the DBMS by LANSA.
- 3. Add Rules. Rules can be added at the field or file level in the repository. For example, a list of values rule can be added which checks that Customer Status is either 'O'-Open or 'C'-Closed.
- 4. Define File Relationships. By defining relationships to other files in the repository, you are changing a group of files into a true application database. File relationship information can be used to help generate your application logic.
- 5. Define Referential Integrity. Rules for referential integrity will ensure that your database is not corrupted. For example, you cannot delete a Customer if active orders exist in the Order file.
- 6. Add Special Features. You might choose to add features such as virtual fields, batch control or predetermined join fields to the file. For example, virtual fields can be created to substring the Customer Name into First Name and Last Name.
- 7. Make the File Operational. Finally, you will create an Object Access Module for the Customer file. The OAM will combine the above information which you have entered into the repository. If changes are made to fields, files or rules, a new OAM must be created.

#### Also See

Creating Fields Creating Application Databases ↑ 2.3.5 Step 1. Building the Repository

# 2.3.6 Step 2. Building the RDML

Once you have completed your repository development, you will begin to build the LANSA RDML for your application. There are three methods which you can use to build RDML.

#### LANSA Modeling & Prototyping Tools

LANSA provides an object-oriented design and prototyping tool which will build a full application prototype based on your object-action model. This tool uses LANSA templates and is fully programmable so that you can control the RDML code generated.

#### LANSA Programmable Templates

Whenever you create a new RDML function, you should use a LANSA template. LANSA templates can develop complete programs or just blocks of code. Templates can be used to generate Universal, Web Optimized and Windows Optimized applications. Because templates are programmable, you can customize them to meet your site standards. You can write a template to create any standard block of code your application requires. (Refer to Application Templates.)

#### **Manually Define**

Finally, you can manually define your RDML code. When manually defining RDML, you can copy RDML code from your existing applications. You can also execute templates from within the RDML editor.

Your RDML can be developed using the LANSA for i or Visual LANSA development environment. If you are building a Universal user interface style application, the RDML code will be exactly the same in both the IBM i and Windows environment. For example, you can develop code for the IBM i from the Windows environment.

If you are creating Windows Optimized applications, the components must be created using Visual LANSA. (The application definitions can be stored on the IBM i but will not execute in the IBM i environment.)

#### Also See

Creating Applications using Functions Creating Applications using Forms ↑ 2.3 Application Development

# **RDML & Application Types**

Before you begin to code your RDML, you must know the type of application or user interface style you require. Again, LANSA allows you to create applications using the:

- 2.1.3 Universal Model
- 2.1.4 Windows Optimized Model
- 2.1.5 Web Optimized Model

Your business will often begin 2.1.8 Mixing Models to meet different needs based on the 2.1.7 Types of Business Applications required.

While a common set of LANSA Repository skills are required, it is important to remember that specialized RDML development skills will be required depending upon the application style. Development using component requires knowledge of RDMLX commands as well as an understanding of Properties, Events and Methods.

 $\uparrow$  2.3.6 Step 2. Building the RDML

# 2.4 Multilingual Applications

If you are developing an application that may be used in more than one country or that requires more than one language, you will need to be familiar with LANSA's multilingual features. The decision to use multilingual development must be made very early in the project. Review the following topics:

2.4.1 Multilingual Support

2.4.2 Multilingual Partitions

2.4.3 What can be Multilingual?

2.4.4 Multilingual Objects Descriptions

2.4.5 Multilingual Considerations

It is recommended that you open and review the Multilingual Application Design Guide.

 $\$  2. Creating LANSA Applications

# 2.4.1 Multilingual Support

Before attempting to use multilingual support in LANSA you should clearly understand whether or not you will need to use multilingual support.

You must use multilingual support if any of the following conditions apply:

- You want to produce applications that use a bi-directional language such as Hebrew or Arabic.
- You want to produce applications that use ideographic characters (or double byte character set characters) such as Chinese, Japanese or Korean.
- You want to produce an application that end users can execute in more than one language, i.e. the same application will appear in more than one language.
- You do NOT need to use multilingual support to produce applications that execute in Italian, and ONLY in Italian. To produce applications that execute in Italian (and only in Italian), you just need to configure the national language portions of your normal LANSA system to work in the Italian language. However, to produce applications that can execute in Italian, French and German, you do need to use multilingual support.

Note: The ability to use national language capabilities in a normal LANSA system (without using multilingual support) only applies to languages that use the Latin alphabet (or a derivation from it). For applications in bi-directional or IGC/DBCS languages you must use multilingual support, regardless of whether or not the resulting applications are truly multilingual (i.e. able to operate in more than one language).

#### Also See

Multilingual Application Design Guide

# 2.4.2 Multilingual Partitions

If you are creating a multilingual application, you will start by defining a partition or LANSA development environment which supports multilingual development. The partition definition screen has an option "Multilingual support" which can be specified as either YES or NO to indicate whether or not multilingual support is required in this partition. If you specify YES for this option, you must also specify YES for the SAA/CUA standards option.

When creating a new partition, if you choose the option to copy system fields from another partition, only the non-multilingual field information will be copied since no languages have yet been defined for this partition. You should use the export and import facility of LANSA to set up the multilingual information later.

Note: Until a default language has been defined for a multilingual partition, the partition will not be considered to be multilingual.

By defining a LANSA environment which supports the development of multilingual applications, you are making it possible to create a single application which will execute in more than one language.

Note: If you are unsure whether or not your application will require multilingual support, you are advised to select multilingual support as YES. You can easily remove or change a partition from multilingual to non-multilingual, but you can not change a non-multilingual partition to multilingual.

#### Also See

Multilingual Application Design Guide

# 2.4.3 What can be Multilingual?

Once the partition multilingual attributes have been specified, the multilingual object definitions can be added. Multilingual definitions are supported for any object in LANSA that has a description.

A multilingual definition means that the object description can be entered for each language.

It is important not to confuse multilingual definitions with data. A file may have its description in more than one language. For example, a field like Department description would have the following definitions:

GermanAbteillungs-BezeichnungFrenchDescription DepartementEnglishDepartment Description

The appropriate multilingual description will appear on the data entry panel for a German, French and English user. The actual data entered into the file will be based on the user's input. Data is not translated by LANSA.

#### Also See

Multilingual Application Design Guide

# 2.4.4 Multilingual Objects Descriptions

LANSA allows multilingual definitions to be specified when objects are created or edited. The multilingual attributes for objects are entered as part of the object definition.

Typically, when the object is created, a description is entered based on the default development language. This default description will be used for all languages defined in the partition. You must edit the object descriptions to enter the appropriate translation for each language.

For example, when a field such as EMPNO is created, an English description of "Employee Number" is entered. The field is then opened in the editor to add field labels, field descriptions and field column headings for each language defined in the partition.

If you are planning to develop multilingual applications, please be sure to review the *LANSA Multilingual Application Design Guide* in detail.

### Also See

Multilingual Application Design Guide

# 2.4.5 Multilingual Considerations

Following are some important considerations for multilingual applications:

- The multilingual facilities can only be activated in partitions that use the SAA/CUA standards option.
- When defining a new partition (or modifying an existing partition) to support multilingual applications a "default" language must be nominated. The partition will not be considered multilingual until this language has been defined.
- It is virtually impossible to fully develop a DBCS application on a machine that does not have IBM i DBCS support, (i.e. a DBCS primary language) and have DBCS capable devices attached.
- For applications in bi-directional or IGC/DBCS languages you must use multilingual support, regardless of whether or not the resulting applications are truly multilingual (i.e. able to operate in more than one language).
- When specifying multilingual error message text, use multilingual variables to specify the text. This way the textual content of the message will be determined at execution time and thus issued in the correct language.
- It is recommended that you use error message numbers from user message files rather than straight message text for all messages.
- When setting up a partition for multilingual support, you must use care with exporting / importing. For example, you cannot import a multilingual export into a non-multilingual partition. Special steps must be performed. Review the Multilingual Application Design Guide in detail.

#### Also See

# **3. Creating Fields**

The field definitions form the foundation of the LANSA Repository. Since the field definition impacts files, screens, reports, functions and more, it is very important to properly define the fields in the repository.

- 3.1 Getting Started with Repository Fields
- 3.2 Developing with Fields
- 3.3 Field Rules and Triggers Development
- 3.4 Repository Help Text Development
- 3.5 System & Multilingual Variables Development
- 3.6 Field Visualization Development

#### Also See

Creating Application with Components

# **3.1 Getting Started with Repository Fields**

Before starting to create new fields in the repository, review the following topics:

- 3.1.1 Modeling Fields and Files
- 3.1.2 Field Naming Standards
- 3.1.3 Corporate Data Dictionary Concept
- 3.1.4 Before You Begin Checklist
- ↑ 3. Creating Fields

# 3.1.1 Modeling Fields and Files

The LANSA development environment supports modeling tools to assist in the rapid application design and development of your application. Data modeling tools will build both field definitions and file definitions. (Refer to Using Modeling Tools.)

If you are building a new application or re-engineering an application, you should use a modeling tool to help accelerate the development of your application.

Modeling tools can help enforce site standards for the fields, files, processes and functions that are used for your application development.

1 3.1 Getting Started with Repository Fields

# **3.1.2 Field Naming Standards**

Standards are needed to protect your investment in the repository. The following field naming recommendations are provided for "in house" application development, (as opposed to vendor "package" development).

Fields should be in the format xxxxxiii where:

- A field name should be 9 characters or less and should not contain an '#' or '@' symbol.
- The xxxxx part of the name should be 1 to 6 characters and as meaningful as possible given the size limitation. For example: PARTNO- Part Number, CUSTNO- Customer Number, SALQTY- Sales Quantity
- The iii is optional and should be used for "working" or temporary versions of the original xxxxx field in RDML functions. The iii suffix should be standardized and only a limited number of variations allowed. For example: WRK- working value of the field VIR- virtual field definition TOT- totaled value of the field
- All iii suffix versions of the field should be defined in the repository.

Also note the following recommendations:

- File definitions should only contain xxxxx field names (with the exception of virtual fields and predetermined join fields).
- Do not define fields in RDML functions, even when the reference field option is used. This greatly strengthens the cross-referencing power of LANSA. Use the reference field option wherever possible.

An additional advantage of the above naming standards is that when a developer attempts to change the repository and specifies field name xxxxxx, all the iii variations will also appear due to LANSA's generic search and display facility.

While you may not choose to follow the samples provided, we strongly recommend that you enforce a set of standards for field names at your site.

Please review the LANSA Object Names in the LANSA Technical Reference *Guide*.

### Also See

LANSA Application Design Guide

 $\Uparrow$  3.1 Getting Started with Repository Fields

# 3.1.3 Corporate Data Dictionary Concept

The concept of a corporate data dictionary evolved from integrated development environments in which a single computer processing department supports all information systems requirements for a company. Briefly, a corporate data dictionary means that information is defined in terms of the corporate or complete company view and not just a single application's view of data.

For example, how often have you found that the same piece of data has been given many different descriptions and definitions throughout the application system. A Customer Number may be described as Customer ID, Customer Number, ID Number Customer, Cust ID, Customer No, or Cust ID No. Do these refer to the same data or different pieces of data?

Committing to a corporate data dictionary concept will involve:

- Defining objects in one place and one place only.
- Referring all developers to this place for definitions.
- Defining elements using "wide vision" because the object is for corporate use and not just one application. A corporate dictionary means objects are reusable.
- Creating and following definition standards.
- Providing security to control who can change and use the data dictionary information.
- More work during the initial design phase of a project.

The use of the corporate data dictionary concept will produce long-term benefits as it extends the life of your applications and makes integration of your applications possible.

To enforce the corporate data dictionary concept, you may want to limit the developers who are allowed maintain fields in the repository. (Refer to Controlling Repository Maintenance and Development Environment Administration.)

↑ 3.1 Getting Started with Repository Fields

# 3.1.4 Before You Begin Checklist

Before you begin you development of fields in the Repository, it is recommended that the following checklist be reviewed:

- The purpose, concepts and architecture of the LANSA Repository are understood so that your development efforts will properly centralize application information.
- All repository development procedures, standards and guidelines are known by/available to every member of the project team. (Refer to Controlling Repository Maintenance and Development Environment Administration.)
- LANSA Object naming standards and guidelines are known by/available to every member of the project team. (Refer to Field Naming Standards and the LANSA Object Names in the LANSA Technical Reference Guide.)
- All field level repository features have been identified and development impacts are understood. (Refer to 3.2.1 How are Repository Fields Used?)
- A decision has been made regarding the use of the 3.1.3 Corporate Data Dictionary Concept.
- You have read the 3.2 Developing with Fields section of this guide and you have identified the methods that will be used to create fields in the repository.
- 1 3.1 Getting Started with Repository Fields

## **3.2 Developing with Fields**

As you begin to create new fields in the Repository, you should be familiar with the following topics:

- 3.2.1 How are Repository Fields Used?
- 3.2.2 How to Create New Fields
- 3.2.3 Steps for Completing Fields in Repository
- 3.2.4 Copying Versus Reference Fields
- **3.2.5 Reference Fields**
- 3.2.6 System Fields
- 3.2.7 Field Considerations

#### Also See

- 3.1.4 Before You Begin Checklist
- ↑ 3. Creating Fields

## 3.2.1 How are Repository Fields Used?

The repository field definitions are the foundation of your business application. The objects within the repository are often built off these basic definitions. Your components and the functions will be built using the field definitions so it is very important to take time to carefully define fields in the repository. It is the centralization of a single definition and its reuse that provides the great productivity benefits.

Properly defining all field characteristics in the repository is critical. When you define a field's characteristics, you are really defining screen layouts, report layouts and other important information which will be used in your applications. For example, a default value defined in the repository will be automatically used when a screen is displayed; or a function can ask for a field to be set to its default, based on the repository field value. It is important to take the time to carefully consider each characteristic of the field definition.

Remember, the LANSA Repository is an Active Object Repository. (Refer to What is an Active Repository?) For example, a prompt function can be assigned to fields in the repository. The specified prompt function is automatically available from each screen on which the field appears. When the prompt function is changed in the repository, the new function is immediately available on all screens in the application. There is no recompilation of the application.

If you are using LANSA's Multilingual Support Facility (MSF), you will enter descriptions, labels and column headings for all languages specified for the partition. As noted earlier, MSF allows a single application to be executed in more than one language at the same time. (Refer to Multilingual Applications and System & Multilingual Variables Development.)

Also, remember that the LANSA Repository is "open" and the information can be shared and reused. Other PC-based application development tools can access the repository field definitions and use this information when creating and executing their applications. (Refer to Open Model .)

1 3.2 Developing with Fields

## 3.2.2 How to Create New Fields

There are several ways to create a field definition in the repository:

- 1. Manually define a field
- 2. Copy an existing field
- 3. Manually define a field using an existing field as a reference
- 4. Load an existing externally defined file into the Repository
- 5. Use a Modeling tool.

**Option 1**. You may manually create all the details of the field definition. If you are not using the modeler or loading existing files, then all your fields will be created manually.

**Option 2**. You may create a new field by copying an existing field. Copying an existing field allows you to copy the basic field characteristics as well as the rules and help text.

**Option 3**. You may manually create the field but use a Reference Field to define the field's characteristics. Using a reference field is similar to a copy, except it permanently links two field definitions together. Changes made to the referenced field will affect any fields that reference it. (Refer to 3.2.4 Copying Versus Reference Fields.)

**Option 4**. If you are building from existing applications, you can load your existing file or table definitions into LANSA. The file load utilities will create field definitions for the fields in the loaded files.

**Option 5.** You may use the Visual LANSA Modeling tool to automatically build the fields required for the file definitions. (Refer to Using a Modeling Tool.)

### Also See

3.2.3 Steps for Completing Fields in Repository

1 3.2 Developing with Fields

## 3.2.3 Steps for Completing Fields in Repository

The basic development steps to build your fields in the LANSA Repository will depend upon the approach used to create the initial field definitions.

These approaches have been described:

Manually Define New Fields Use Modeling Tools

Load Other Files

#### **Manually Define New Fields**

You may create a new field in the repository by manually defining the field or by copying an existing field. You may use a Reference Field to create the new field. As you create a new field, you should complete the following steps:

- 1. Use proper naming standards when the field is created. These are described in the 3.1.3 Corporate Data Dictionary Concept.
- 2. Set all required field characteristics based on your site standards. (Refer to 3.2.7 Field Considerations.)
- 3. Define field level help text for the user and the developer. (Refer to 3.4 Repository Help Text Development.)
- 4. Create field visualizations if required for component or WAM development. (Refer to 3.6 Field Visualization Development.)
- 5. Optional: In special cases, field level rules and triggers can be added. (Refer to 3.6 Field Visualization Development.)
- 6. Test your field definitions by using a simple test function. (For example, you can use the FLDTEST application template.)

### **Use Modeling Tools**

If you have used a modeling tool to create the database, the basic field definitions are automatically created for you. Most field definitions will use Reference Field based on attribute data types that are part of the modeling tool. You will complete the following steps to update field definitions:

- 1. Open each field created by the modeling tool.
- 2. Review and update changeable field characteristics where necessary. (Refer to Field Considerations.)
- 3. If help was not added during the modeling phase, define field level help text

for the user and the developer. (Refer to Repository Help Text Development.)

- 4. Create field visualizations if required for component or WAM development. (Refer to Field Visualization Development.)
- 5. Optional: In special cases, field level rules and triggers can be added. (Refer to Field Rules and Triggers Development.)
- 6. Test your field definitions by using a simple test function or by using an application prototype.

### Load Other Files

If you have loaded other files to use an existing database, all fields in the files are automatically added to the repository. You will complete the following steps to update field definitions:

- 1. Review file definitions to identify created fields in the Repository.
- 2. Open each field created when the file was loaded. Review and update the field characteristics. (Refer to Field Considerations.)
- 3. Define field level help text for the user and the developer. (Refer to Repository Help Text Development.)
- 4. Create field visualizations if required for component or WAM development. (Refer to Field Visualization Development.)
- 5. Optional: In special cases, field level rules and triggers can be added. (Refer to Field Rules and Triggers Development.)
- 6. Test your field definitions by using a simple test function.

### Also See

3.2.5 Reference Fields

1 3.2 Developing with Fields

# **3.2.4 Copying Versus Reference Fields**

It is important to understand the difference between copying a field in the repository and using a reference field as part of a field definition.

Copying a field creates a one-time duplication of the field characteristics. The field definition is copied as it exists at the time of creation. You are allowed to copy help text, validation rules, triggers and visualizations. When there is a change to the original field, there is no impact to the copied field.

Reference fields create a permanent link between fields. Field characteristics are permanently inherited so that when the reference field is changed, all fields referencing the changed field are also automatically updated. Once a field has been defined using a reference field, you are not be allowed to change specific inherited characteristics. You are allowed to import help text, validation rules and triggers from the reference field at any time.

A reference field does not adopt the visualizations of the referenced field.

**Important Note:** If you add a new rule to a reference field or change help text of a reference field, the new rule/help text is NOT automatically added to all referencing fields because the copying if these details is optional.

#### Also See

- 3.2.5 Reference Fields
- 1 3.2 Developing with Fields

## 3.2.5 Reference Fields

A reference field is an example of inheritance in the repository. When defining a field in the repository, you can specify that it has a reference field. The reference field establishes a permanent link between the two fields. Once defined, a field will have its characteristics based on the reference field, that is, they will be inherited from the reference field. When the definition of a reference field is changed, this immediately changes the following characteristics of all fields that reference the changed field:

- Type
- Edit code
- Length
- Edit word
- Decimal positions
- Input & output attributes
- Default value

As long as a field is not created from another referenced field, any field can be used as a reference field.

Once a field is defined as having a reference field, LANSA allows you to selectively copy rules, triggers and help text from the referenced field. You can copy this information when the field is created and you can also selectively copy these details after the field has been created.

A reference field does not adopt the visualizations of the referenced field.

If a prompt function is specified for a reference field, the prompt function will be available for all fields which refer to the reference field, provided the field does not have its own prompt function defined.

Reference fields are commonly used by the LANSA modeling tools to establish attribute data types or standard fields in a model. If you have used a LANSA modeling tools to create your database, your fields will probably use reference fields.

#### **Reference field example**

Reference fields are commonly used for dates. Most applications have dates: Start Date, Birth Date, Application Date, Closing Date, Delivery Date and so on. If all the dates reference a field called DATE, then the specific validations and error messages relating to the date can be copied from the DATE field. If all the dates are to be within, say, 60 days of the current date except Birth Date, then you create, for Birth Date, the specific range validations required. Should the date field change in size, such as the YY of the year to YYYY, then only the DATE field needs to be changed. (Naturally, report and screen changes resulting from such a change need to be handled in the traditional manner.)

**Note:** You cannot delete a field while references are made to it. An error message will be displayed if you attempt to delete the field.

For extra security, reference fields are sometimes given a 3.2.6 System Fields designation to prevent them from being deleted while no other fields refer to them.

### Also See

3.2 Developing with Fields

## 3.2.6 System Fields

Any field in the repository can be nominated as a system field.

System fields have one special property:

• You cannot delete a system field. An error message will be displayed if you attempt to delete the field. The field must be changed to a non-system field before you can delete it.

LANSA is shipped with some standard System fields (named STD\_xxxx) and these are added to each new partition that you create.

**Note:** Any field can be moved from one partition to another by using the LANSA export/import facilities. The system field designation simplifies the process of creating standard fields in a new partition.

 $\uparrow$  3.2 Developing with Fields

## 3.2.7 Field Considerations

Following are some important considerations when defining fields:

- Set site standards for field Descriptions, Labels and Column Headings. (Refer to Corporate Data Dictionary Concept.)
- Always assign a default value for a field. These values are used on screens, reports and for database operations.
- Field input and output attributes are very powerful. Input attributes influence screen behavior. Output attributes will have database impacts. For example, stamping attributes can automatically assign values to field values when fields are updated in a databases.
- Match the field's *Column Heading* text length to the field length. For example, a 2 character state code might have a 2 character column heading of "ST" instead of using "STATE CODE".
- Remember that single line column headings require less display area on screens and reports.
- If the partition is multilingual, the description specified for the default partition language will be used as the default for other languages. You must manually edit and update other languages after initially creating the field.
- It is recommended that most validations rules be specified at file level in the repository. (Refer to Field versus File Level Rules.)
- Remember to add field level help text. It is entered just once in the repository and then it becomes available to every LANSA application that uses that field. Some simple text can have a huge impact across your application. (Refer to Repository Help Text Development.)
- If you are using Reference Field, use impact analysis to review the list of related fields before making changes to the referenced field.

Remember, when you define a field's characteristics, you are really defining screen layouts, report layouts and other important information which will be used in your applications. Spending time to properly set up the repository will save significant time during application development.

1 3.2 Developing with Fields

## 3.3 Field Rules and Triggers Development

LANSA allows rules and triggers to be defined at the field level in the repository. As an introduction to rules and triggers, review the following topics:

3.3.1 Types of Field Validation Rules

3.3.2 Validation Rule Levels

3.3.3 Field Rules/Triggers and the Object Access Module

3.3.4 Field versus File Level Rules

Before adding field rules and triggers, review File Rules and Triggers Development.

### Also See

Rules and Triggers Definitions in *Technical Reference Guide*. Field Rules and Triggers Tab in the *User Guide*.

↑ 3. Creating Fields

# **3.3.1 Types of Field Validation Rules**

In LANSA, the applications business rules or validation checks are centrally defined in the repository at either field or file level. Centralized field validations remove the need for developers to code the same validation into each function that uses the same field. The same types of rule checks are provided for both fields and files.

It is important to understand that field validation rules are performed as part of database file operations. A rule is checked when a file operation (such as an record insert, update or delete) is performed using the specified field.

LANSA provides six types of validation checks in the Repository:

### **Range of Values:**

Will check if a field is within a given range

Example: check if Employee Number is between A0000 to A9999.

#### List of Values:

Allows a field to be checked against a list of values

Example: check if the employee's skill Grade is P, F, M or D.

#### File Entry Lookup:

Allows a field to be "looked up" in a code table or file

Example: check if a Section code exists in the Section file.

### **Evaluate an Expression:**

Allows a series of conditions to be evaluated to check a field Example: check if Department code is blank.

### Call a User Program:

Allows field checking by your validation program or LANSA function Example: call an existing program which validates the day of the week.

### Date Format/Range:

Allows a field to be validated as a date in a certain format (DDMMYYCC, CCYYMMDD etc.) and in a certain range

Example: check that an employee's start date is entered in DDMMYY format and is not in the future.

You can often perform the same check using different rules. For example, to test if a field is blank, you could use a List of Values check or an Evaluate an

Expression check.

### Also See

Field Rules and Triggers Development

# 3.3.2 Validation Rule Levels

Within LANSA, there are three levels at which business rules can be defined. The levels of the validation hierarchy include:

- Field Level (defined as part of the repository field definition)
- File Level (defined as part of the repository file definition)
- Function/Component Level (defined using RDML).

The order in which validation rules are performed at execution time is Function/Component Level, then Field Level and lastly File Level. Your programs may perform RDML coded checks before they perform database operations. The field and file validation rules are not called until the program performs a database operation using the Object Access Module (OAM).

The order of importance of the validation hierarchy is defined as Field, then File and lastly Function/Component. For a field to be used in a file, it must be defined to the repository first. If the same field is used in three different files, all three files will all refer to the same repository field. Consequently, each file will use the same field level validation rules for that specific field. (This logic is also true of multiple programs which use one file.) Hence, field rules are included in files which are then used by functions/components. Even though a function/component level check may be performed before the field and file checks, the field and file checks are considered higher in the hierarchy because they provide ultimate control of your validations.

Almost all validations will be specified at file level. (Refer to Field versus File Level Rules.)

↑ 3.3 Field Rules and Triggers Development

## 3.3.3 Field Rules/Triggers and the Object Access Module

The field (and file) level validation rules which you have defined in the repository are implemented in the Object Access Module, (also referred to as the I/O Module). The repository validation rules cannot be tested until an Object Access Module (OAM) has been created. For example, a data entry screen may include the Employee Number field. This field on the screen has no validation rules associated with it until the field is used to perform a file operation such as adding a new record to the Employee Master file. When the insert to the database is performed, the Employee Number validation checks in the OAM are executed and error message will be sent back to the screen.

When you have added or changed any of the validation rules in the repository, it is necessary to make the file operational again. The OAM protects the functions from changes to rules so that the functions do not have to be recompiled. When you change a field level rule, you must use impact analysis tools to determine which files, if any, are affected by the rule change. You must manually request that the appropriate Object Access Modules be rebuilt. LANSA will not detect this change. If changes are made to rules a the file level, these changes are detected by LANSA and will cause the OAM to rebuilt.

1 3.3 Field Rules and Triggers Development

## 3.3.4 Field versus File Level Rules

Field level rules and triggers will apply to every file that uses a specific field. In the Personnel System example, the DEPTMENT field is used in the Department (DEPTAB) and Employee (PSLMST) file. If a field level rule is defined so that DEPTMENT cannot be blank, then this rule applies to both the DEPTAB and PSLMST files. This field level rule will means that the Department code cannot be blank in the Department File and it cannot be blank in the Employee Master file. If Department code is allowed to be blank in Employee Master file, you cannot use a field level rule. You need to define the rule to the Department file only by using a file level rule.

You must be careful that the field level rule or trigger is truly a "global" rule to an application. Global means that the rule or trigger always applies to a specific field, no matter what file the field is used in.

It is recommended that most validations rules be specified at file level. File level validations rules allow you to control the context of a field. Also, if you have multiple validation rules for a single field, you are better able to control the order that rules are processed if all rules are at the same level. Finally, any changes to file level validation rules are automatically detected and the OAM will be rebuilt. You must use impact analysis to determine how a field level rule change will affect the database and you must manually rebuild OAMs for affected files. (Refer to Field Rules/Triggers and the Object Access Module.)

Before adding field rules and triggers, review File Rules and Triggers Development.

↑ 3.3 Field Rules and Triggers Development

# **3.4 Repository Help Text Development**

LANSA supports a full compliment of help text for the application and provides some powerful automatic help text generation facilities. With LANSA, providing application help text becomes a natural part of application development.

Help text is provided at the field, component, process and function levels.

- 3.4.1 Types of Help Text
- 3.4.2 How Help Text Can Be Used
- 3.4.3 How to Create Field Help Text
- 3.4.4 How to Create Process/Function/Form Help Text
- 3.4.5 Repository Help Editor
- 3.4.6 How to Use Special Characters
- 3.4.7 How to Use Special Values
- 1 3. Creating Fields

# 3.4.1 Types of Help Text

Following IBM's SAA/CUA guidelines, help text should be available within all applications for any selection field, selection list or entry field. Help text provides quick and easy access to information for users. Adding help text to LANSA applications is a very easy task since LANSA automatically controls handling of the Help key. Because help text can be added or changed without any recompiling, you can have the users add or modify help text and it is available immediately. Help text is another example of the benefits of LANSA's Active Repository.

LANSA supports different types of help text:

- Field Help in the *Technical Reference Guide*.
- Component Help in the *Technical Reference Guide*.
- **Process Help** in the *Technical Reference Guide*.
- Function Help in the *Technical Reference Guide*.

Again, all of these levels are automatically handled by LANSA and are cursor sensitive, i.e. the cursor location will determine which help text is displayed. LANSA includes a help index to allow users to locate and display both process and function level help.

LANSA also supports both end-user and technical help text. End-user help will be available to all end-users running applications; however, technical help is only displayed to developers. Using *LANSA for i, LANSA Documentor* will use both technical and end-user help text to create both the technical and user guides for your system.

If the LANSA Modeling tools were used, the system analyst's notes are automatically transferred to the appropriate field, process and/or function level help text.

## **3.4.2 How Help Text Can Be Used**

Once help text is defined in the repository, it is automatically made available to your LANSA applications. For example, if help text is created for the Department field, this help text can be accessed from any screen or panel which has the Department field displayed. The field can be used for input or output. The user simply positions the cursor on the field and presses the help key. The active repository will generate and display the help text in the appropriate execution language (if a multilingual partition is being used).

The LANSA application can be IBM i-based (executing as a 5250 application) or the LANSA application can execute on the PC as a standalone or client/server application. The application might also be executed over the Internet as a Web application.

If you have created Windows applications with other development tools, these applications can access the LANSA help text. A PC application can display the LANSA help text in a message box simply by calling a single LANSA API (application program interface). The same help text is now available to the application. This is another example of sharing information in the LANSA Repository.

Help text is an important part of the repository definitions. Whenever you create a field, process, function, or form, you should immediately define help text for the item.

Even if you are not planning to build a LANSA-based application, it is important to remember that the help text can still be used by developers as documentation for the fields in the repository. For example, technical help text will help document the field definitions. (Refer to \$\$TECH in How to Use Special Values.)

## 3.4.3 How to Create Field Help Text

As noted, LANSA can automatically generate field level help text in the correct language for the user. The help text is derived from the repository and includes the field definition information, as well as information regarding the validation rules. Automatic help text generation can be disabled at either the system or the field level.

When editing a field definition, you can create help text by using the Repository Help Tab in the *User Guide*. The text is entered directly using the capabilities of the Repository Help Editor. If the field uses a reference field, you can import the help text from the reference field.

You should also be familiar with How to Use Special Characters and How to Use Special Values in help text.

### 3.4.4 How to Create Process/Function/Form Help Text

LANSA automatically controls the handling of the Help key for process, function and form level help. Again, the help for LANSA applications is context sensitive. LANSA determines whether field, function or form level help is required and automatically displays the corresponding text.

Unlike field level help which can be generated by LANSA, the process, function and form level help must be entered by the developer. Help text should be created for each process, as well as for each function within a process. Each Form has repository level help as well as code level help inside the component.

When editing a process definition, function or form, you can create help text by using the Process Repository Help Tab in the *User Guide*.

The text is entered directly using the capabilities of the Repository Help Editor. You should also be familiar with How to Use Special Characters and How to Use Special Values in help text.

# **3.4.5 Repository Help Editor**

Visual LANSA provides a very simple line editor for creating you help text. This line editor does not control fonts or font presentation characteristics (bold, italic, underline, etc.).

LANSA HELP text has the following characteristics:

- It is free format. No restrictions usually exist on the content or format of HELP text.
- Help text may uses special Help Text Enhancement & Substitution Values. (Refer to How to Use Special Values.)

Please note the following about the Repository Help Editor:

- The editor is a simple line editor. It is not a document editor.
- The editor supports a maximum line length of 74 characters.
- The editor does not support any line wrapping. Once the 74 character limit is reached, you must manually move text to the next line.
- The editor supports basic cut, paste and copy features. For example, you could copy text from a Word document and paste it into the LANSA Help editor.
- You can only paste 74 characters of text per line into the editor. If you try to paste a large amount of text into the editor an error may result. You must check that the text being pasted has been broken into lines of no more than 74 characters by adding a line feed character at the end of each "line" of text.
- Text will appear as it is entered into the editor unless Special Characters or values are used. (Refer to How to Use Special Characters.)
- In a multilingual partition, the Visual LANSA Repository Help Tab uses a separate tab for each language. The special \$\$LANGUAGE tags are not used to designate languages. Always remember to review all language tabs when editing the help text.

**Note:** A simple line editor approach is required in order to allow text to be mapped exactly for different database and presentation requirements. Each line of help text is assigned an internal line number.

## **3.4.6 How to Use Special Characters**

Some special features are provided by LANSA to enhance the display of HELP text. These consist of special characters that can be used as follows :

#### Special Character Effect

% (percentage)	Display in high intensity
{ (left parenthesis)	Underline
@ (at)	Display in reverse image
~ (accent)	Blink
} (right parenthesis)	Revert to normal display mode
\(backslash)	Revert to normal display mode

**Note:** These special characters may be changed when not appropriate in your language.

For example: The following text:

The {underlined field} indicates an input

would appear as:

The underlined field indicates an input

Notice that the "%" and "{" are replaced by blanks. Where multiple enhancement characters are used together with no spaces between them they are "consolidated" into one enhancement character. For instance "%{~" (high intensity, underline, blink) would be consolidated into one enhancement character that is used in the same position as the "~".

Note that a display enhancement cannot be continued over multiple lines. Display enhancements are considered to end at the end of the text line, even if a "revert to normal display" character ("\" or "}") is not used:

contact the{Accounts department or your supervisor for more information.

would be displayed as:

contact the Accounts department or your supervisor for more information.

#### Also See

Substitution/Control Values in the Technical Reference Guide Substitution/Control Values - Visual LANSA Only in the Technical Reference Guide Help Text Attributes in the Technical Reference Guide 3.4.7 How to Use Special Values

1 3.4 Repository Help Text Development

## 3.4.7 How to Use Special Values

Some special values are also provided by LANSA to control how the Help text is displayed. These special values all begin with \$\$. Following are a few examples:

Special Value	Effect
\$\$LANGUAGE	Indicates the language of the help text.
\$\$FLDNAM	Inserts the name of the field.
\$\$NOAUTO	Indicates that automatic help text should not be generated.
\$\$USER	Designates end-user help text.
\$\$TECH	Designates technical help text.
\$\$RIGHT	Causes help text to be right aligned when displayed.
\$\$PAGEC	Causes page number to be inserted into the text.

For example, you can enter user and technical help as follows:

\$\$USER
This text is displayed to an end-user when the application executes.
\$\$TECH
This text is not displayed to the end-user. It is displayed to developers only.

For a complete list of values, refer to the *Technical Reference Guide*:

- Substitution/Control Values
- Substitution/Control Values Visual LANSA Only
- Help Text Attributes.

#### Also See

3.4.6 How to Use Special Characters

1 3.4 Repository Help Text Development

## **3.5 System & Multilingual Variables Development**

System variables can be used in a number of places such as default values for a field in the repository, in validation rules or as parameters in RDML commands.

System Variables are system wide and exist in all partitions. If a system variable is changed in one partition, this change impacts all partitions.

You may manually create your own system variables within LANSA. Before creating your own System Variables, you should review the following topics:

3.5.1 System Variable Concepts

Create System Variables in the Visual LANSA User Guide.

3.5.2 Multilingual Text Variable Concepts

Create Multilingual Variables in the Visual LANSA User Guide.

If you create your own system variables, information for creating evaluation programs is provided in System Variable Evaluation Programs in the *Visual LANSA Developer Guide*.

LANSA includes a set of *Special Variables* that perform special functions such as storing the next available numbers in a sequence. For more about the *Special Variables* and the other shipped system variables, refer to the *Technical Reference Guide*:

General Variables Function Only Variables Special Variables Built-In Function Variables ↑ 3. Creating Fields

# **3.5.1 System Variable Concepts**

A system variable is a global variable that is used across all LANSA partitions. A system variable is used to store commonly used and often variable or dynamic, pieces of information.

For instance, the current date is a commonly used piece of information that varies every day. It could be assigned the system variable named \*TODAYS\_DATE.

Once created, this system variable name can be used in many places within LANSA:

- As a field's default value in the repository.
- In any of the 6 standard types of field validation rules.
- Just about anywhere in the RDML components of LANSA.

The extensive use of system variables in validation rules has many advantages:

- The validation rules are simpler and easier to understand.
- The validation rules are more powerful and more accurate.
- The validation rules last longer and are easier to maintain.

For instance, you could have a field named DATDUE that is only to be the current date. The required validation rule could be done in one of these ways:

- As a range of values rule. Validate DATDUE as being in range \*TODAYS\_DATE to \*TODAYS\_DATE.
- As a list of values rule. Validate DATEDUE as being in the list \*TODAYS\_DATE.
- As a simple logic rule. Validate DATDUE by evaluating the expression '#DATDUE = \*TODAYS\_DATE'.

While many system variables are provided with the shipped version of LANSA, their primary use is to aid in accessing your own company's information. For example, you could create the following system variables to match your company's specific requirements:

System Variable	Description	Туре	Len	Dec
*PERIOD	Current financial period	Ν	4	0
*LASTPERIOD	Last financial period	Ν	4	0
*NEXTPERIOD	Next financial period	Ν	4	0

*BEGINMONTH	Date of first day of current month	Ν	6	0
*ENDMONTH	Date of last day of current month	Ν	6	0
*SYDOFFICE	Sydney office name and address code	А	7	
*MEBOFFICE	Melbourne office name and address code	А	7	
*COMPANY	Company number for use in G/Ledger	А	2	

Note that the system variables listed above are not supplied with LANSA. They are suggestions for creating your own system variables.

When a system variable is defined to LANSA, the *Method of Derivation* must be specified. This indicates when the system variable is to be derived or evaluated. It can be specified as STATIC or DYNAMIC. In addition, the *Set Value by Calling* parameter identifies the name of a LANSA function or 3GL program that will "evaluate" the system variable. It is this function/ program that is important because it provides the "intelligence" required to determine the correct value to place in the system variable.

Note: System variables exist at the LANSA system level and are shared by all partitions. If the value of a system variable is changed in one LANSA partition, then all partitions will now use this new value.

1 3.5 System & Multilingual Variables Development

# **3.5.2 Multilingual Text Variable Concepts**

Like system variables, multilingual text variables can be used in a number of places, such as default values for a field in the repository, in validation rules or as parameters in RDML commands.

Unlike system variables, multilingual text variables are specific to a partition.

The following example of a text string means the same thing in different languages, but has a different textual value in each language:

"Locate Employee by Name" "Ricerca Alfabetica Impiegati" "Zoek Werknemer op Naam"

If this was a piece of text that you wished to place on a screen panel, a report, or send in a message, then a multilingual variable (sometimes called an \*MTXT variable) should be used.

By positioning an \*MTXT defaulted field on the screen, rather than just text, the values that appear on the screen panel at execution time will vary according to the language being used.

In many respects \*MTXT variables are just like system variables (described in 3.5.1 System Variable Concepts), and they can be used in the same places as system variables:

- As a field's default value in the repository.
- In any of the 6 standard types of field validation rules.
- Just about anywhere in the RDML components of LANSA.

The extensive use of multilingual variables in validation rules has many advantages:

- The validation rules are simpler and easier to understand.
- The validation rules are more powerful and more accurate.
- The validation rules work in multiple languages.

For example, imagine that you have created these multilingual variables:

*MTXTMON	"Monday" in all desired languages
*MTXTTUE	"Tuesday" in all desired languages
*MTXTWED	"Wednesday" in all desired languages

*MTXTTHU	"Thursday" in all desired languages
*MTXTFRI	"Friday" in all desired languages
*MTXTSAT	"Saturday" in all desired languages
*MTXTSUN	"Sunday" in all desired languages
*MTXTERR001	"Day of week specified is invalid" in desired languages

Then, in just one list of values rule in the repository, a file or an RDML program, you could validate the day of the week (in the current language) and issue an error message (in the current language).

Additionally, as the list of \*MTXT variables grows, they will become used repeatedly in different applications. For instance \*MTXTMON, \*MTXTTUE, etc might be used as column headings on screen panels or reports.

**Note:** Before attempting to make large scale use of \*MTXT variables, please develop and use naming standards for them.

#### Also see

Create Multilingual Variables

1 3.5 System & Multilingual Variables Development

## 3.6 Field Visualization Development

By default all fields in the repository are visualized as entry fields when you add them to a component.

Typically you have only one visualization for a field, but it is possible to have several visualizations which are used in different contexts.

The component definition does not affect the field definition: in other words all the field attributes, rules and triggers remain unchanged.

3.6.1 Visualization Types

3.6.2 Defining Field Visualizations

3.6.3 Adding a Visualization

3.6.4 Field with Several Visualizations

3.6.5 Using a Field with Visualization

3.6.6 Using Field Visualization in a List

3.6.7 Differences between Field Visualization on a Form and in a List component

3.6.8 Returning the Value from a Field Visualization

3.6.9 Form Samples

3.6.10 List Samples

3.6.11 Miscellaneous

3.6.12 Field Visualization in Depth

3.6.13 Dynamic Visualizations

#### Also See

Field Visualization Tab in the User Guide.

↑ 3. Creating Fields

## **3.6.1 Visualization Types**

There are several visualization types you can choose for a field. Which one is most appropriate depends on the type of field and the type of values it can have. The types available follow:

## **Edit Box**

When you add a field to a component, by default it is visualized as an edit box:

Employee Number	A2301	

You can change the default length, width and other characteristics of the edit box. The same is true for all visualizations.

## **Edit Box with a Prompter**

You can define a prompter component for a field. The prompter is a userdefined reusable part. When the field has a prompter, a button with ellipses is added to the edit box so the user can display the prompter:

Employee Number	A2301	

At run-time when the prompter button is clicked, the prompter component is displayed to allow the user to choose a value.

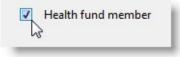
## Set of Radio Buttons

When the field can have one of a limited set of values, it can be displayed as radio buttons:



## **Check Box**

When a field's value can be True or False, it can be defined as a check box:



#### List

When a field can have one of a set of values, it can be visualized as a list or a drop-down:



#### **Spin Edit Box**

A numeric field can be visualized as a spin edit box:

Amount:	1204	÷
		13

#### **Track Bar**

Numeric fields can be visualized as a track bar where appropriate:



## **Progress Bar**

In some instances you may want to visualize a numeric field as a progress bar:



## **DateTime Picker and Calendar**

Datetime fields can be visualized as a drop-down datetime picker or a calendar:

1/06/	1987	K				
44 -	•	Ju	ne 19	987		• ••
Mon	Tue	Wed	Thu	Fri	Sat	Sun
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5
		Toda	y: 29	9/10	/201	12

## **Reusable Part**

You can define a reusable part to for visualizing the field.

## When Embedded in Lists

Embedded in a list (Grid, ListView, TreeView) fields can be displayed as:

- Check Box
- Image
- ImageAndText
- Edit
- ComboBox
- MultiLineEdit

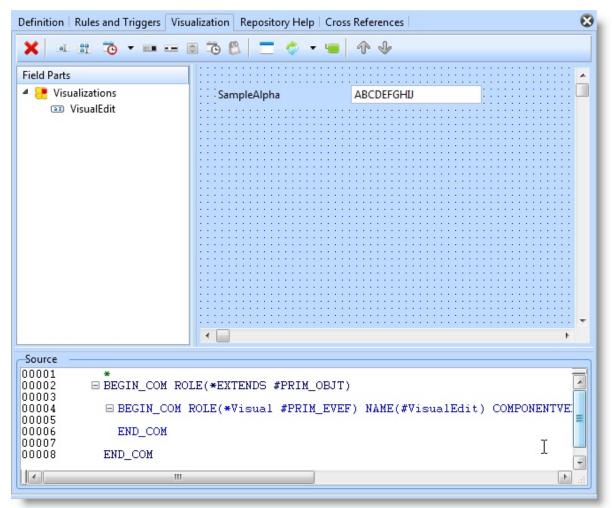
Embedded in a list (Grid, ListView, TreeView) fields can be edited as:

- Check Box
- Image
- ImageAndText
- Edit
- ComboBox
- MultiLineEdit
- SpinEdit
- ReusablePart

↑ 3.6 Field Visualization Development

# 3.6.2 Defining Field Visualizations

Field visualizations are defined in the Visualization tab in the field definition:



The default visualization is an edit box. The source code window shows the statements defining the visualization.

The Role( \*Visual #PRIM\_EVEF) specifies that the field is displayed as an edit box.

#### Also See

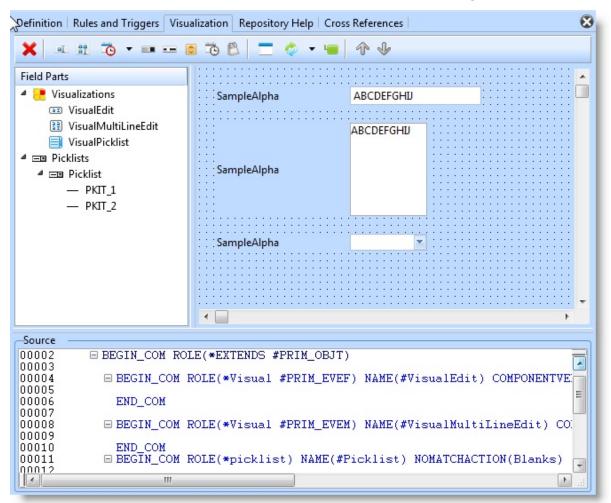
Field Visualization Tab in the User Guide

1 3.6 Field Visualization Development

## 3.6.3 Adding a Visualization

To add another visualization for the field, use the toolbar on the tab.

When you select one of the visualizations, it is added to the field's component definition. The new visualization becomes visible in the design view:



The component defined for the Role property defines the type of visualization. These are the Roles for the available visualizations:

Edit box	#PRIM_EVEF
Spin edit field	#PRIM_EVSE
Track bar	#PRIM_EVTB
Progress bar	#PRIM_EVPB
Radio buttons, check box, lists (PickList visualizations)	#PRIM_EVPL

If you define several visualizations for a field and if you want to make one of them the default visualization of the field when it is added to another component, set the DefaultVisual property of the visualization to True: In the following picture the track bar visualization is made the default:

Details				
VisualTrackBar	*			
E • ?				
Properties Events N	/lethods			
Caption				
Cursor	*NULL			
🖉 DefaultVisual	True			
🖻 DragStyle	None			
Ellipses	None			
🖻 Enabled	True			
iller Height	21			
🖻 Hint				

Only one of the field's visualizations can be the default visualization.

You need to save the updated definition for the field before you can use the visualizations you have defined.

↑ 3.6 Field Visualization Development

## 3.6.4 Field with Several Visualizations

Typically you define just one visualization for a field, but it is possible to define several visualizations and use them in different contexts:

Definition Rules and Triggers Visualization Repository Help Cross References					
X = # * • •	🛚 🙃 🖺 📄 🚍 🧇 🕶 🖷	Ŷ ∳			
Field Parts		****			
<ul> <li>Visualizations</li> <li>VisualEdit</li> </ul>	SampleNumeric	123456789012345			
<ul> <li>WisualTrackBar</li> <li>WisualProgress</li> </ul>	SampleNumeric				
🗟 VisualSpinEdit	SampleNumeric				
	SampleNumeric	123456789012345			
	•				

Here is the source for the visualizations:

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

\* Edit Box Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Defaultvisual(True) Usepicklist(False) Componentversion(1) Height(21) Width(358) End\_Com

\* Track Bar Begin\_Com Role(\*Visual #PRIM\_EVTB) Name(#VisualTrackBar) Height(21) Width(237) End\_Com \* Progress Bar Begin\_Com Role(\*Visual #PRIM\_EVPB) Name(#VisualProgress) Height(21) Width(237) End\_Com \* Spin Edit Begin\_Com Role(\*Visual #PRIM\_EVSE) Name(#VisualSpinEdit) Usepicklist(False) Componentversion(1) Height(21) Width(358) End\_Com

End\_Com

1 3.6 Field Visualization Development

## 3.6.5 Using a Field with Visualization

When you add a field to a component, its default visualization is displayed. Here is the #SIZE field which is a List on a form at design-time:

•	•					•	•		•		•	•				•		•	•	•	
•	5	)iz	e	:													-	1	•	•	
							•	•	•	•	•	•	•		•	•	•	•			
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	i

This is its definition on the form:

Define\_Com Class(#SIZE.Visual) Name(#SIZE) Displayposition(1) Left(32) F

With a Field with Several Visualizations you can use a visualization which is not the default by specifying the name of the visualization like this:

#FieldName.VisualizationName

For example this form contains three visualizations for afield:

SampleAlpha	ABCDEFGHU	:
Size:	<b>-</b>	••••••
SampleAlpha	ABCDEFGHIJ	
		••••••

Here are the component definitions on the form:

DEFINE\_COM CLASS(#SampleAlpha.VisualEdit) NAME(#SampleAlpha) DISPLAYPOSITION(1) LEFT(16) PARENT(#COM\_OWNER) TABPOSITION(1) TOP(8)

DEFINE\_COM CLASS(#SampleAlpha.VisualMultilineEdit) NAME(#EDIT) DISPLAYPOSITION(2) LEFT(24) MARGINLEFT(150) PARENT(#COM\_OWNER) TABPOSITION(2) TOP(88) WIDTH(289)

#### DEFINE\_COM CLASS(#SampleAlpha.VisualPicklist) NAME(#RADIO) DISPLAYPOSITION(3) HEIGHT(40) LEFT(24) MARGINLEFT(50) PARENT(#COM\_OWNER) TABPOSITION(3) TOP(40) WIDTH(145)

The DropDownList which is the default visualization is simply defined as #SIZE.Visual whereas the other visualizations are specified by name.

1 3.6 Field Visualization Development

# 3.6.6 Using Field Visualization in a List

Using a Field Visualization in a list is relatively complex as it requires a greater understanding of the parts available for visualizing fields in different ways.

On a form you have a direct one to one relation ship between a field class and that field class visualized on a form. This is not true for Field Visualization in a LANSA List component. Grid, ListView and TreeView all support Field Visualization capabilities. These components work by adding columns to the List component. So where on a form we use the FieldName.ClassName standard for accessing visualizations in a List Component we use the Source property of the column and then manipulate various properties of the column.

# **Basic Example**

Create a Form and add a Grid with 1 column for #STD\_NUM.

Change the DisplayAppearance and EditAppearance properties of the column to reflect the visualization required. Eg.For column sourced to #STD\_NUM change the EditAppearance property to SpinEdit. Also ensure ReadOnly property is FALSE for Column.

Define\_Com Class(#PRIM\_GDCL) Name(#GDCL\_1) Displayposition(1) Edit

Populate the grid from the form's CreateInstance event.

Define\_Com Class(#STD\_NUM.VisualSpinEdit) Name(#STD\_NUM) Display

Compile and run the form. Column is edited as a SpinEdit.

# **3.6.7 Differences between Field Visualization on a Form and in a List component**

- For List components you do not necessarily need to open the Field component source.
- When used on a form a field visualization will always have a corresponding class in the Field class list.
- When used in List components visualizations are chosen through the properties of the column (eg DisplayAppearance, EditAppearance).

The concept of field visualization in list type components can get somewhat confusing as the full set of Field Visualization options are available on a column regardless of the field being sourced to that column. This can easily lead to incorrect results for inexperienced users. (eg. It is possible via a Columns properties to try and Visualize a field as an ImageSet even though you may not have defined the available images for the field by adding a picklist and items in the field source).

To avoid the pitfalls of using Field Visualization in the LANSA List components it is advised that you take the provided sample that is the closest match to your requirements as a starting point. This will then be easy to modify.

#### 3.6.10 List Samples

1 3.6 Field Visualization Development

## 3.6.8 Returning the Value from a Field Visualization

Visualizations such as edit box, spin edit box, track bar and progress bar return the value specified in the visualization as usual.

However, visualizations based on picklists return the value specified in the Value parameter of the selected picklist item.

This form shows how the Example Spin Edit Box and the example List (based on picklist) return their value:

	]					x
	Employee Salary Size	6,000.00 Medium	÷.	Value of #Salary: Value of #Size: Item Caption of #Size:	0006000 Medium Medium	
L						

Here is the form source:

```
Function Options(*DIRECT)
BEGIN_COM ROLE(*EXTENDS #PRIM_FORM) CLIENTHEIGHT(189)
CLIENTWIDTH(637) HEIGHT(227) LEFT(437) TOP(222) WIDTH(653)
DEFINE COM CLASS(#SIZE.Visual) NAME(#SIZE)
DISPLAYPOSITION(2) LEFT(24) MARGINLEFT(100)
PARENT(#COM_OWNER) TABPOSITION(2) TOP(64) WIDTH(201)
DEFINE COM CLASS(#STD TEXTS.Visual) NAME(#STD TXT 1)
CAPTION('Value of #Size:') DISPLAYPOSITION(1) HEIGHT(19)
LABELTYPE(Caption) LEFT(256) MARGINLEFT(120)
PARENT(#COM OWNER) TABPOSITION(1) TOP(72)
USEPICKLIST(False) WIDTH(217)
DEFINE COM CLASS(#STD TEXTS.Visual) NAME(#STD TXT 2)
CAPTION('Item Caption of #Size:') DISPLAYPOSITION(3) HEIGHT(19)
LABELTYPE(Caption) LEFT(256) MARGINLEFT(120)
PARENT(#COM OWNER) TABPOSITION(3) TOP(96)
USEPICKLIST(False) WIDTH(217)
DEFINE COM CLASS(#SALARY.Visual) NAME(#SALARY)
DISPLAYPOSITION(4) LEFT(24) MARGINLEFT(100)
PARENT(#COM OWNER) TABPOSITION(4) TOP(24) WIDTH(201)
```

DEFINE\_COM CLASS(#STD\_COUNT.Visual) NAME(#STD\_COUNT) CAPTION('Value of #Salary:') DISPLAYPOSITION(5) HEIGHT(19) LABELTYPE(Caption) LEFT(256) MARGINLEFT(120) PARENT(#COM\_OWNER) TABPOSITION(5) TOP(28) USEPICKLIST(False) WIDTH(217)

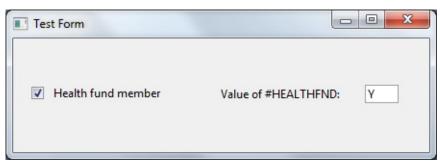
Evtroutine Handling(#SALARY.Changed) Options(\*NOCLEARMESSAGES \*NOCLEARERRORS) Change Field(#STD\_COUNT) To(#SALARY) Endroutine

Evtroutine Handling(#SIZE.Changed) Options(\*NOCLEARMESSAGES \*NOCLEARERRORS) Change Field(#STD\_TXT\_1) To(#SIZE) Change Field(#STD\_TXT\_2) To('#SIZE.text') Endroutine

End\_Com

A check box visualization returns the value of the first picklist item if the check box is not ticked, and the value of the second picklist item when the check box is ticked.

This form shows the value of the example Check Box:



Here is the source for the form:

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Caption('Test Form') Height(1 Define\_Com Class(#HEALTHFND.Visual) Name(#HEALTHFND) Displaypo Define\_Com Class(#STD\_TEXTS.Visual) Name(#STD\_TEXTS) Caption('Val

Evtroutine Handling(#HEALTHFND.Changed) Options(\*NOCLEARMESSA Change Field(#STD\_TEXTS) To(#HEALTHFND) Endroutine

End\_Com

1 3.6 Field Visualization Development

## 3.6.9 Form Samples

This section shows how field visualizations can be used on forms and reusable parts:

Edit Box	ComboBox	ButtonSet
Edit Box with Prompter	CheckBox	Reusable Part
Spin Edit	Image	DateTime Picker
TrackBar	ImageAndText	Calendar
ProgressBar	ListBox	

1 3.6 Field Visualization Development

#### **Edit Box**

## **Field Name**

EMPNO

#### Description

How to visualize a field as an Edit Box on a form. Field Source

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Defaultvisual(T

 $End\_Com$ 

 $End\_Com$ 

Form Source

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#EMPNO.Visual) Name(#EMPNO) Displayposition(1) Let

 $End\_Com$ 

## Appearance

Employee Number	

↑ 3.6.9 Form Samples

## **Edit Box with Prompter**

## Field Name

EMPNO

## Description

How to visualize a field as an Edit Box on a form and have a prompter for the edit box.

You can add a prompter to a standard visualization. The first step to this is to create the prompter. The prompter form MUST conform to the following in order to work as expected.

- Prompter Form must have a default property
- Prompter Form must contain some mechanism for returning a Modalresult(ok) when closed successfully (usually done through an OK button)

## How to create the sample

- Create a form called EMPNOPRMT. Paste the prompter code from below
- Open EMPNO field in the component editor and paste Field source from below.
- Create a form and copy the form code from below. Compile and run.

## How it works

When creating a prompter form we ensure we follow the prompter form guidelines. We add a property to the form as follows and ensure it is made the default property for the form (eg Defaultpty(p\_Employee\_number) )

Define\_Pty Name(p\_Employee\_number) Get(\*auto #empno) Set(set\_empno)

The set of this property will be called when the user activates the prompter (via either F4 or the ellipses button). When called the property will be given the current value of the field. This is to allow the prompter form an opportunity to ensure that the correct employee is selected prior to the display of the form. The form is then shown modally and the user will select the new employee. Next the user chooses the OK button and the form is unrealized. The ok button handler sets the modalresult for the form to ok ( Modalresult(ok) ) to inform the calling field that all was successful and a new employee was chosen. If the ModalResult is not set the field will presume the prompt was cancelled and not

try to retrieve a new value.

The field then calls the get of the default property to retrieve the prompter value and updates its contents from the new value.

#### **Prompter Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Defaultpty(p\_Employee\_num)

Define\_Com Class(#PRIM\_GPBX) Name(#GPBX\_2) Caption('All Employees Define\_Com Class(#PRIM\_LTVW) Name(#LISTVIEW) Displayposition(1) E Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Caption('First Name') Ca Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Caption('Last Name ') C Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Parent(#LISTVIEW) So

Define\_Com Class(#PRIM\_ATLM) Name(#LAYOUT1) Define\_Com Class(#PRIM\_ATLI) Name(#IL301) Attachment(Center) Manage

Define\_Com Class(#PRIM\_SPLM) Name(#LAYOUT2) Orientation(Vertical) Define\_Com Class(#PRIM\_SPLI) Name(#IS102) Manage(#GPBX\_2) Parent(;

Define\_Com Class(#PRIM\_PMNU) Name(#PMNU\_1)

Define\_Com Class(#PRIM\_MITM) Name(#HSPLIT) Caption('Split Form Hot Define\_Com Class(#PRIM\_MITM) Name(#VSPLIT) Caption('Split Form Ver Define\_Com Class(#PRIM\_PANL) Name(#PANL\_1) Displayposition(2) Heig Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Bottom) Man Define\_Com Class(#PRIM\_PHBN) Name(#PB\_OK) Buttondefault(True) Cap Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_2) Buttoncancel(True) Cap

Group\_By Name(#TREEITEMS) Fields(#DEPTMENT #DEPTDESC #SECT]

Define\_Pty Name(p\_Employee\_number) Get(\*auto #empno) Set(set\_empno)

Ptyroutine Name(set\_empno) Define\_Map For(\*input) Class(#empno) Name(#employee)

Selectlist Named(#LISTVIEW)

Continue If('#Employee \*ne #empno')

Set Com(#listview.currentitem) Focus(true) Selected(true)

Endselect

Endroutine

Evtroutine Handling(#com\_owner.CreateInstance) Set Com(#com\_owner) Caption('Employee Browser/Selector')

\* Change Field(#TREEITEMS) To(\*NULL)

\* Select Fields(#DEPTMENT #SECTION #EMPNO #SURNAME #GIVENA Select Fields(#LISTVIEW) From\_File(PSLMST) Fetch Fields(#DEPTDESC) From\_File(DEPTAB) With\_Key(#DEPTMENT) F Fetch Fields(#SECDESC) From\_File(SECTAB) With\_Key(#DEPTMENT #SF Use Builtin(BCONCAT) With\_Args(#GIVENAME #SURNAME) To\_Get(#FU

Add\_Entry To\_List(#LISTVIEW) Set Com(#ListView.currentitem) Image(#vi\_employ) Endselect

Endroutine

Evtroutine Handling(#VSPLIT.Click) Set Com(#layout2) Orientation(vertical) Set Com(#VSplit) Enabled(False) Set Com(#HSplit) Enabled(True) Endroutine

Evtroutine Handling(#HSPLIT.Click) Set Com(#layout2) Orientation(horizontal) Set Com(#HSplit) Enabled(False) Set Com(#VSplit) Enabled(True) Endroutine

Evtroutine Handling(#PB\_OK.Click)

Invoke Method(#COM\_OWNER.CLOSEFORM)

Set Com(#com\_owner) Modalresult(ok)

Endroutine End\_Com

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#ENTRYFIELD) Defaultvisu

 $End\_Com$ 

Begin\_Com Role(\*prompter #EMPNOPRMT) Name(#PROMPTER) Defaultp

 $End\_Com$ 

 $End\_Com$ 

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#EMPNO.Visual) Name(#EMPNO) Displayposition(1) Let

 $End\_Com$ 

## Appearance

E	
Employee Number	 ]

All Employees			
First Name	Last Name		
Veronica	Brown		
Fred	Bloggs		
Fred	Smithson		
Anne	Simpson		
Shirley	Jones		
John	Smythe		
Robert	Smithe		
Ruth	Smithson		
Peter	Smiths		
Jack	Smithers		
George	Snell		
Alison	Sneddon		
Damian	Snashall		
Kate	Perry		
Christopher	Perron		
Patrick	Paul		
George	Dattincon		-
		OK Cancel	

#### Note:

• In the OK button click handler the following line is used to inform the calling field that the prompter is being closed successfully and that a new value has been chosen.

Set Com(#com\_owner) Modalresult(ok)

If the above line was not present, the calling field would presume the prompter form was cancelled and was not successful so would not try to retrieve a new value for the field.

- After adding a prompter form to a field by adding a \*Prompter class in the field source you still do not automatically get the ellipses for the prompter when you use the field on a form. To make the ellipses visible set the ShowPrompter property of the field to TRUE.
- 1 3.6.9 Form Samples

## Spin Edit

## **Field Name**

STD\_NUM

## Description

How to visualize a field as a SpinEdit Box on a form.

Through the field component source add a spin edit visualization. Save the field and then use the spinedit visualization on a form.

## **Field Source**

\*

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Height(19) Usej

End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVSE) Name(#VisualSpinEdit) Defaultvisu

End\_Com End\_Com

## Form Source

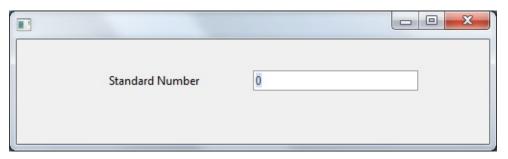
Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#STD\_NUM.Visual) Name(#STD\_NUM) Displayposition(

 $End\_Com$ 

## Appearance



**Note:** The SpinEdit visualization name is VisualSpinEdit. It also has DefaultVisual(TRUE) set. This means that the SpinEdit can be accessed on a form either of the following ways:

Define\_Com Class(#STD\_NUM.Visual) Name(#STD\_NUM) Displayposition( or

Define\_Com Class(#STD\_NUM.VisualSpinEdit) Name(#STD\_NUM) Display ↑ 3.6.9 Form Samples

## TrackBar

## **Field Name**

VOLUME(Signed2,0)

#### Description

How to visualize a field as a TrackBar on a form.

Create a field called VOLUME which is signed and has a length of 2. Through the field component source remove the PRIM\_EVEF visualization and add a TrackBar visualization. Save the field and then use the visualization on a form.

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVTB) Name(#VisualTrackBar) Defaultvis

 $End\_Com$ 

End\_Com

#### Form Source

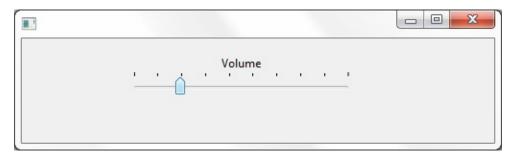
Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#VOLUME.Visual) Name(#VOLUME) Displayposition(1)

 $End\_Com$ 

Appearance



# 1 3.6.9 Form Samples

## **ProgressBar**

## **Field Name**

PROGRESS(Signed2,0)

## Description

How to visualize a field as a ProgressBar on a form.

Create a field called PROGRESS which is signed and has a length of 2. Through the field component source remove the PRIM\_EVEF visualization and add a ProgressBar visualization. Save the field and then use the visualization on a form.

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVPB) Name(#VisualProgress) Defaultvisu

End\_Com

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(130) Clientwidth

Define\_Com Class(#PROGRESS.Visual) Name(#PROGRESS) Displaypositio Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_INC) Caption('Increment') Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_DEC) Caption('Decrement

Evtroutine Handling(#PHBN\_INC.Click)

Change Field(#STD\_IDNO) To('#PROGRESS + 1') Change Field(#PROGRESS) To(#STD\_IDNO)

Endroutine

Evtroutine Handling(#PHBN\_DEC.Click)

Change Field(#STD\_IDNO) To('#PROGRESS - 1') Change Field(#PROGRESS) To(#STD\_IDNO)

Endroutine

End\_Com

# Appearance

Progre	ess bar	
Increment	Decrement	

↑ 3.6.9 Form Samples

#### ComboBox

## **Field Name**

SEX (Alpha 1)

## Description

How to visualize a field as a ComboBox on a form.

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add PicklistItems for male and Female to the Picklist and change the Appearance property of the PRIM\_EVPL to DropDownList. Save the field and use the VisualPicklist class on a form.

## **Field Source**

BEGIN\_COM ROLE(\*EXTENDS #PRIM\_OBJT)

BEGIN\_COM ROLE(\*Visual #PRIM\_EVEF) NAME(#VisualEdit) COMPONENTVERSION(1) HEIGHT(19) USEPICKLIST(False) WIDTH(178)

End\_Com

BEGIN\_COM ROLE(\*picklist) NAME(#LIST) DEFINE\_COM CLASS(#PRIM\_PKIT) NAME(#ITEM1) CAPTION('Male') DEFAULT(True) PARENT(#LIST) VALUE('M') DEFINE\_COM CLASS(#PRIM\_PKIT) NAME(#ITEM2) CAPTION('Female') PARENT(#LIST) VALUE('F') End\_Com

BEGIN\_COM ROLE(\*Visual #PRIM\_EVPL) NAME(#VisualPicklist) APPEARANCE(DropDownList) COMPONENTVERSION(1) DEFAULTVISUAL(True) HEIGHT(69) MARGINLEFT(50) WIDTH(125)

End\_Com

END\_COM End\_Com

#### **Form Source**

Function Options(\*DIRECT)

BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) CLIENTHEIGHT(131) CLIENTWIDTH(319) HEIGHT(169) LEFT(440) TOP(196) WIDTH(335) DEFINE\_COM CLASS(#SEX.Visual) NAME(#SEX) DISPLAYPOSITION(1) LEFT(64) PARENT(#COM\_OWNER) TABPOSITION(1) TOP(24)

End\_Com

#### Appearance

Sex	Male 💌	
J. J	Male	
	Female	

↑ 3.6.9 Form Samples

## CheckBox

## **Field Name**

ACTIVE (Alpha 1)

## Description

How to visualize a field as a Check Box on a form.

#### Uses

This Visualization type is useful for a Field that is used as a Boolean(TRUE\FALSE).

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add two picklist items (Yes and No) to the Picklist and change the Appearance property of the PRIM\_EVPL to CheckBox. Save the field and use the VisualPicklist class on a form.

## **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('Yes') Default(Tru-Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('No') Parent(#Pick

End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

End\_Com

End\_Com

## Form Source

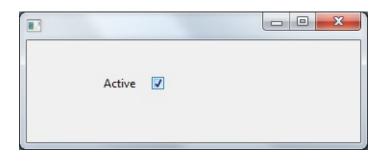
Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(130) Clientwidth

Define\_Com Class(#ACTIVE.VisualPickList) Name(#ACTIVE) Displaypositi

End\_Com

## Appearance



#### Note:

Clicking the check box will set the corresponding value for the field from the picklist items. Values are chosen from the following rule:

- Checkbox Unchecked = First picklist item value
- CheckBox Checked = Last PicklistItem value
- CheckBox grayed = Any picklistItem between first and last or an invalid value. (eg Changing the value of the field to a value that is not the first or last item in the picklist will result in the check box being grayed.)

1 3.6.9 Form Samples

#### Image

# Field Name

NATION (Alpha 3)

## Description

How to visualize a field as an Image set on a form.

This Visualization type is useful to provide a more visual means of selecting values from a list.

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add picklist items to represent nationalities and attach an image to each picklist item. Change the Appearance property of the PRIM\_EVPL to Image. Save the field and use the VisualPicklist class on a form.

## **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('USA') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Japan') Image(#V

Define\_Com Class(#PRIM\_PKIT) Name(#Item) Caption('Great Britain') Imag Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Australia') Image End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

 $End\_Com$ 

End\_Com

## **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(130) Clientwidth

Define\_Com Class(#NATION.VisualPickList) Name(#NATION) Displayposit

End\_Com

## Appearance



1 3.6.9 Form Samples

## ImageAndText

## **Field Name**

NATION (Alpha 3)

How to visualize a field as an Image set on a form.

This Visualization type is useful to provide a more visual means of selecting values from a list.

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add picklist items to represent nationalities and attach an image to each picklist item. Change the Appearance property of the PRIM\_EVPL to ImageAndText. Save the field and use the VisualPicklist class on a form.

## **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('USA') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Japan') Image(#V

Define\_Com Class(#PRIM\_PKIT) Name(#Item) Caption('Great Britain') Imag Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Australia') Image End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

 $End\_Com$ 

End\_Com

## Form Source

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(130) Clientwidth

Define\_Com Class(#NATION.VisualPickList) Name(#NATION) Displayposit

End\_Com

# Appearance



↑ 3.6.9 Form Samples

## ListBox

# **Field Name**

NATION (Alpha 3)

## Description

How to visualize a field as a ListBox on a form.

#### Uses

This Visualization type is useful to provide user with a list of values to choose from. Also useful when the list is too large to display all options in a combo box.

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add picklist items to represent nationalities and attach an image to each picklist item. Change the Appearance property of the PRIM\_EVPL to ListBox. Save the field and use the VisualPicklist class on a form.

## **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('USA') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Japan') Image(#V

Define\_Com Class(#PRIM\_PKIT) Name(#Item) Caption('Great Britain') Imag Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Australia') Image End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

End\_Com

End\_Com

## Form Source

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(130) Clientwidth Define\_Com Class(#NATION.VisualPickList) Name(#NATION) Displayposit End\_Com

## Appearance

		x
Nationality	USA Japan Great Britain Australia	

## 1 3.6.9 Form Samples

#### **ButtonSet**

#### **Field Name**

SEX (Alpha 1)

#### Description

How to visualize a field as a set of radio buttons on a form.

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add PicklistItems for male and Female to the Picklist and ensure the Appearance property of the PRIM\_EVPL is ButtonSet. Save the field and use the VisualPicklist class on a form.

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Defaultvisual(T

End\_Com

Begin\_Com Role(\*picklist) Name(#LIST) Define\_Com Class(#PRIM\_PKIT) Name(#ITEM1) Caption('Male') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#ITEM2) Caption('Female') Parent(# End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Height(69) V

 $End\_Com$ 

End\_Com

#### **Form Source**

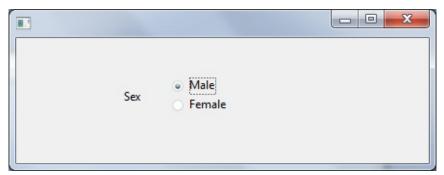
Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(130) Clientwidth

Define\_Com Class(#SEX.VisualPickList) Name(#SEX) Displayposition(1) He

End\_Com

# Appearance



1 3.6.9 Form Samples

#### **Reusable Part**

## **Field Name**

SECTION(Alpha 2)

## Description

How to visualize a field as an ReusablePart on a form.

There are times when you may wish to provide a field as a list of options but the built in PicklistList is not flexible enough for your case. One example of this is where you may want a section field to display a list of sections from a database.

## **Reusable Part Source (SECTIONR)**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_PANL) Defaultpty(Section) Displaypo \* Reusable consists of a combo box with a client layout item attachment Define\_Com Class(#PRIM\_CMBX) Name(#SECT\_CBX) Displayposition(1) Define\_Com Class(#PRIM\_CBCL) Name(#CBCL\_1) Displayposition(1) Pare Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1) Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Center) Mana

Define\_Pty Name(Section) Get(Get\_Section) Set(Set\_Section)

Evtroutine Handling(#COM\_OWNER.Initialize) Options(\*NOCLEARMESS/

\* Select all entries from PSLMST and add to our combo box Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#SECT\_CBX)

Endselect

Endroutine

Ptyroutine Name(Get\_Section) Define\_Map For(\*output) Class(#SECTION) Name(#o\_section)

\* Default property Get.

\*

\* Get\_Entry on the combo box focus item and set the return value

```
If_Ref Com(#SECT_CBX.focusitem) Is_Not(*NULL)
```

Change Field(#STD\_NUM) To('#SECT\_CBX.FOCUSITEM.ENTRY')

Get\_Entry Number(#STD\_NUM)

Endif

Set Com(#o\_section) Value(#SECTION)

Endroutine

Ptyroutine Name(Set\_Section) Define\_Map For(\*input) Class(#SECTION) Name(#i\_section)

\* Default property Set.

\*

\* Select\_List on Combo and set focus to mathing item from input

Selectlist Named(#SECT\_CBX)

If Cond('#SECTION \*ne #I\_SECTION')

Continue

Endif

Set Com(#SECT\_CBX.currentitem) Focus(true) Selected(true)

Endselect

Endroutine

 $End\_Com$ 

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Defaultvisual(Ti

End\_Com

Begin\_Com Role(\*Visual\_Part #SECTIONR) Name(#Combo) Height(19) Wic

End\_Com

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#SECTION.COMBO) Name(#SECT\_CBX) Height(49) Le Define\_Com Class(#PRIM\_LTBX) Name(#LIST) Componentversion(1) Displ Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_SET) Caption('Set Field V Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_GET) Caption('Get Field V Define\_Com Class(#PRIM\_LBCL) Name(#LBCL\_1) Displayposition(1) Parei

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#LIST)

Endselect

Endroutine

Evtroutine Handling(#PHBN\_SET.Click)

If\_Ref Com(#LIST.focusitem) Is\_Not(\*NULL)

Change Field(#STD\_NUM) To('#LIST.FOCUSITEM.ENTRY')

Get\_Entry Number(#STD\_NUM)

Endif

Change Field(#SECT\_CBX) To(#SECTION)

Endroutine

Evtroutine Handling(#PHBN\_GET.Click)

Selectlist Named(#LIST)

If Cond('#SECTION \*ne #SECT\_CBX')

Continue

Endif

Set Com(#LIST.currentitem) Focus(true) Selected(true)

Endselect

Endroutine

 $End\_Com$ 

## Appearance

01 02 03 01 01 03 01 FC *	Set Field Value	FC	*

# ↑ 3.6.9 Form Samples

#### **DateTime Picker**

#### **Field Name**

ADateTime(Datetime, 28)

#### Description

How to visualize a field as DateTime Picker on a form.

You would typically want to visualize datetime fields as a DateTime Picker. This is the default visualization for datetime fields.

#### **Field Source**

BEGIN\_COM ROLE(\*EXTENDS #PRIM\_OBJT)

BEGIN\_COM ROLE(\*Visual #PRIM\_EVDT) NAME(#VisualDateTime) DEFAULTVISUAL(True) HEIGHT(19) WIDTH(370)

END\_COM

END\_COM

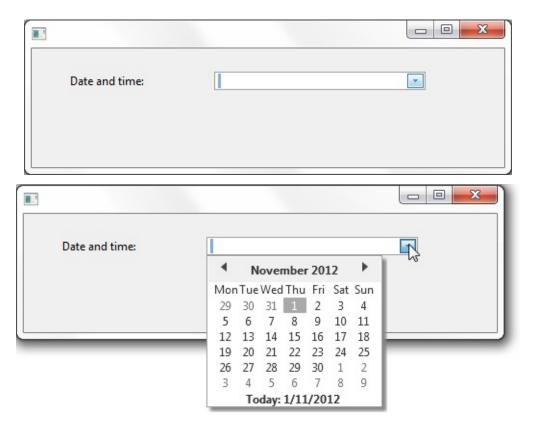
#### **Form Source**

Function Options(\*DIRECT)

BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) CLIENTHEIGHT(123) CLIENTWIDTH(492) HEIGHT(157) LEFT(467) TOP(163) DEFINE\_COM CLASS(#ADATETIME.Visual) NAME(#ADATETIME) DISPLAYPOSITION(1) HEIGHT(19) LEFT(40) PARENT(#COM\_OWNER) TABPOSITION(1) TOP(24) WIDTH(370)

End\_Com

## Appearance



1 3.6.9 Form Samples

## Calendar

## **Field Name**

ADateTime(Datetime, 28)

#### Description

How to visualize a field as a Calendar on a form.

#### **Field Source**

BEGIN\_COM ROLE(\*EXTENDS #PRIM\_OBJT)

BEGIN\_COM ROLE(\*Visual #PRIM\_EVDT) NAME(#VisualDateTime) HEIGHT(19) WIDTH(370)

END\_COM

BEGIN\_COM ROLE(\*Visual #PRIM\_EVMC) NAME(#VisualCalendar) DEFAULTVISUAL(True) HEIGHT(156) MARGINLEFT(183) USEPICKLIST(False) WIDTH(366)

END\_COM END\_COM

#### **Form Source**

Function Options(\*DIRECT)

BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) CLIENTHEIGHT(251) CLIENTWIDTH(492) HEIGHT(278) LEFT(467) TOP(163) DEFINE\_COM CLASS(#ADATETIME.Visual) NAME(#ADATETIME) DISPLAYPOSITION(1) LEFT(40) PARENT(#COM\_OWNER) TABPOSITION(1) TOP(24)

End\_Com

## Appearance

Date:								
	•	N	over	nber	201	2	•	
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
	29	30	31	1		3	4	
	5	6	7	8	9	10	11	
	12	13	14	15	16	17	18	
	19	20	21	22	23	24	25	_
	26	27	28		30	1	2	
	3	4	5	6	7	8	9	

# 1 3.6.9 Form Samples

# 3.6.10 List Samples

This section shows how field visualizations can be used in lists:

#### Edit Box Edit Box with Prompter Spin Edit ComboBox

CheckBox Image ImageAndText Reusable Part

PRIM\_GRID (Grid Control), PRIM\_LTVW (ListView Control) and PRIM\_TRVW (Tree Control) all support Field Visualization. They differ a little in what they allow but if a feature is allowed it can be implemented in the same way as any other control. The following details a little of how these controls differ in their Field Visualization features.

Grid Control	The LANSA Grid offers the most complete feature set for Field Visualization in a List component. Any cell of the grid can be Visualized as per the rules for the column. As the grid offers the full feature set of Field Visualization functions it is with this component that most of the samples to follow will use.
List View Control	<ul> <li>The LANSA List View control also offers the full feature set of Field Visualization but they are subject to the limitations of a ListView. Limitations include:</li> <li>ListView must be have ViewStyle(Report)</li> <li>Only Column with DisplayPosition(1) is editable</li> </ul>
Tree Control	The LANSA Tree Control only allows Edit Appearance capabilities of Field Visualization. This is because a Tree Control must always look like a Tree Control.

Moving on to how Field Visualization is used in the List components: The various columns of the List components PRIM\_GDCL (Grid Column), PRIM\_LVCL (ListView Column) and PRIM\_TVCL (Tree Column) all have properties that support Field Visualization. These properties include:

	<ul> <li>Check Box</li> <li>Edit</li> <li>Image</li> <li>ImageAndText</li> <li>MultiLine Edit</li> </ul>
Edit Appearance	Defines how the item is to be edited. This is distinct from DisplayAppearance as you have a greater choice of options to choose from with EditAppearance. EditAppearances include: • Check Box • Edit • Image • ImageAndText • MultiLine Edit • ReusablePart • SpinEdit
Editor Part	<ul> <li>The name of the component to be used for editing when the EditAppearance is set to ReusablePart. The part entered must conform to the following rule:</li> <li>Reusable Part must have a default property. It is through this property that the current value will be set and get values in the list.</li> </ul>
UsePicklist	This property has been added to PRIM_EVEF, PRIM_EVSE, PRIM_GDCL, PRIM_LVCL & PRIM_TVCL. It is a design time only property. The purpose of this property is to specify whether the picklist of a field, once specified, is to be used as part of the formatting for the relevant component. The primary purpose of this property is seen as a way to introduce Field Visualization concepts bit by bit to a large system. Older components can work as they always have with UsePicklist=FALSE whereas new components can have

	UsePicklist=TRUE to gain access to new functionality.
Editor	The actual Editor used by the column.

Example: A PRIM\_EVSE ( SpinEdit ) has been defined for a numeric field that also includes a Picklist. If the UsePicklist property is FALSE then when visualized the SpinEdit will work as normal through the range of allowable numbers. If UsePicklist=TRUE then the SpinEdit will cycle through the picklist values.

Field Visualization in a List component is somewhat trickier than Field Visualization on a form. Forms are pretty straight forward as you simply add the classes to the field source and then use those classes on a form. You cannot get into too much trouble as you can only access functionality that you have added to your field source. The list components are a little harsher on less experienced users. This is because with a list component all you ever have to work with is a column. This means that the full set of Field Visualizations is always available whether or not they are relevant to the field currently sourced to the column. For this reason greater care should be taken when using Field Visualization in the List components. Ensure you have read all the samples and have a clear understanding of what you are trying to accomplish before proceeding. When in doubt refer to the provided samples for a clue.

1 3.6 Field Visualization Development

#### Edit Box

## Field Name

EMPNO(Alpha 5)

#### Description

How to visualize a field as text or numeric data in a grid.

Create a grid component. From the fields tab add the EMPNO field to the grid as a column. Set the ReadOnly property of the created column to FALSE. Load the grid and the column can be edited.

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VISUALEDIT) Defaultvisu

 $End\_Com$ 

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Displayposition(1) Parent

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

Endroutine

End\_Com

# Appearance

Employee Numbe	
A0070 ···	*

↑ 3.6.10 List Samples

## **Edit Box with Prompter**

# Field Name

EMPNO(Alpha 5)

## Description

How to visualize a field as an Edit Box in a Grid and have a prompter for the edit box.

You can add a prompter to a standard visualization. The first step to this is to create the prompter. The prompter form MUST conform to the following in order to work as expected.

- 1. Prompter Form must have a default property.
- 2. Prompter Form must contain some mechanism for returning a Modalresult(ok) when closed successfully (usually done through an OK button).

## How to create the sample

- 1. Create a form called EMPNOPRMT. Paste the prompter code from below.
- 2. Open EMPNO field in the component editor and paste Field source from below.
- 3. Create a form and copy the form code from below.
- 4. Compile and run.

# How it works:

When creating a prompter form, be sure to follow the prompter form guidelines. Add a property to the form as follows and ensure it is made the default property for the form (eg Defaultpty(p\_Employee\_number))

Define\_Pty Name(p\_Employee\_number) Get(\*auto #empno) Set(set\_empno)

The set of this property will be called when the user activates the prompter (via either F4 or the ellipses button). When called the property will be given the current value of the field. This is to allow the prompter form an opportunity to ensure that the correct employee is selected prior to the display of the form.

The form is then shown modally and the user will select the new employee. Next the user chooses the OK button and the form is unrealized. The ok button handler sets the modalresult for the form to ok ( Modalresult(ok) ) to inform the calling field that all was successful and a new employee was chosen. If the ModalResult is not set the field will presume the prompt was cancelled and not try to retrieve a new value.

The field then calls the get of the default property to retrieve the prompter value and updates its contents from the new value.

#### **Prompter Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Defaultpty(p\_Employee\_num)

Define\_Com Class(#PRIM\_GPBX) Name(#GPBX\_2) Caption('All Employees Define\_Com Class(#PRIM\_LTVW) Name(#LISTVIEW) Displayposition(1) E Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Caption('First Name') Ca Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Caption('Last Name ') C Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Parent(#LISTVIEW) So

Define\_Com Class(#PRIM\_ATLM) Name(#LAYOUT1) Define\_Com Class(#PRIM\_ATLI) Name(#IL301) Attachment(Center) Manage

Define\_Com Class(#PRIM\_SPLM) Name(#LAYOUT2) Orientation(Vertical) Define\_Com Class(#PRIM\_SPLI) Name(#IS102) Manage(#GPBX\_2) Parent(;

Define\_Com Class(#PRIM\_PMNU) Name(#PMNU\_1) Define\_Com Class(#PRIM\_MITM) Name(#HSPLIT) Caption('Split Form Hor Define\_Com Class(#PRIM\_MITM) Name(#VSPLIT) Caption('Split Form Ver Define\_Com Class(#PRIM\_PANL) Name(#PANL\_1) Displayposition(2) Heig Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Bottom) Man Define\_Com Class(#PRIM\_PHBN) Name(#PB\_OK) Buttondefault(True) Cap Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_2) Buttoncancel(True) Cap

Group\_By Name(#TREEITEMS) Fields(#DEPTMENT #DEPTDESC #SECT]

Define\_Pty Name(p\_Employee\_number) Get(\*auto #empno) Set(set\_empno)

Ptyroutine Name(set\_empno) Define\_Map For(\*input) Class(#empno) Name(#employee)

Selectlist Named(#LISTVIEW)

Continue If('#Employee \*ne #empno')

Set Com(#listview.currentitem) Focus(true) Selected(true)

Endselect

Endroutine

Evtroutine Handling(#com\_owner.CreateInstance) Set Com(#com\_owner) Caption('Employee Browser/Selector')

\* Change Field(#TREEITEMS) To(\*NULL)

\* Select Fields(#DEPTMENT #SECTION #EMPNO #SURNAME #GIVENA Select Fields(#LISTVIEW) From\_File(PSLMST) Fetch Fields(#DEPTDESC) From\_File(DEPTAB) With\_Key(#DEPTMENT) F Fetch Fields(#SECDESC) From\_File(SECTAB) With\_Key(#DEPTMENT #SF Use Builtin(BCONCAT) With\_Args(#GIVENAME #SURNAME) To\_Get(#FU

Add\_Entry To\_List(#LISTVIEW) Set Com(#ListView.currentitem) Image(#vi\_employ) Endselect

Endroutine

Evtroutine Handling(#VSPLIT.Click) Set Com(#layout2) Orientation(vertical) Set Com(#VSplit) Enabled(False) Set Com(#HSplit) Enabled(True) Endroutine

Evtroutine Handling(#HSPLIT.Click) Set Com(#layout2) Orientation(horizontal) Set Com(#HSplit) Enabled(False) Set Com(#VSplit) Enabled(True) Endroutine

Evtroutine Handling(#PB\_OK.Click)

Invoke Method(#COM\_OWNER.CLOSEFORM)

Set Com(#com\_owner) Modalresult(ok)

Endroutine End\_Com

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#ENTRYFIELD) Defaultvist

 $End\_Com$ 

Begin\_Com Role(\*prompter #EMPNOPRMT) Name(#PROMPTER) Defaultp

 $End\_Com$ 

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Displayposition(1) Parent

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

#### Endroutine

#### End\_Com

#### Appearance

Employee Numbe			
A0070			
2			
2			
2			
		*	
Employee Browser/Selector			
All Employees			
First Name	Last Name		
Veronica	Brown		-
Fred	Bloggs		
Fred	Smithson		
Anne	Simpson		
Shirley	Jones		
lohn	Smythe		
Robert	Smithe		
Ruth	Smithson		
Peter	Smiths		
lack	Smithers		
George	Snell		
Alison	Sneddon		
Damian	Snashall		
Kate	Perry		
Christopher	Perron		
Patrick	Paul		
George	Pattinson		
John	Moore		
Bradley	Woods		
Jack	Turner		
-			

#### Note:

In the OK button click handler the following line is used to inform the calling field that the prompter is being closed successfully and that a new value has been chosen.

Set Com(#com\_owner) Modalresult(ok)

If the above line was not present then the calling field would presume the prompter form was cancelled and was not successful so would not try to retrieve a new value for the field.

↑ 3.6.10 List Samples

#### Spin Edit

#### **Field Name**

STD\_NUM

## Description

Display the field STD\_NUM in a grid and have the grid editable as a SpinEdit. Create a grid component. From the fields tab add the STD\_NUM field to the grid as a column. Set the ReadOnly property of the created column to FALSE and the EditAppearance property to SpinEdit. Load the grid and the column can be edited as a spin edit

#### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Height(19) Usej

End\_Com

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Displayposition(1) Editap

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

Endroutine

End\_Com

#### Appearance

Standard Number	
0	A
2	
4 🗢	
0	
0	
0	
0	*

#### Note:

It is not necessary to have a PRIM\_EVSE (SpinEdit) visualization in the Field component source in order to edit the field in a grid as a spin edit. It is through the Grid column properties rather than the field component source that the spin edit visualization is used.

↑ 3.6.10 List Samples

#### ComboBox

## **Field Name**

SEX (Alpha 1)

## Description

How to visualize a field as a ComboBox in a grid.

- 1. Through the Field Component source add a Picklist visualization ( PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add PicklistItems for male and Female to the Picklist.
- 2. Save the field.
- 3. Create a grid component. From the fields tab add the SEX field to the grid as a column. Set the following properties on the column:

```
ReadOnly = FALSE.
DisplayAppearance=Edit
EditAppearance=Edit
UsePicklist = TRUE.
```

3. Load the grid and the column can be edited as a drop down combo.

# **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Defaultvisual(T

End\_Com

Begin\_Com Role(\*picklist) Name(#LIST) Define\_Com Class(#PRIM\_PKIT) Name(#ITEM1) Caption('Male') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#ITEM2) Caption('Female') Parent(# End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

End\_Com

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Displayappearance(Edit) ]

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

Endroutine

End\_Com

#### Appearance

Sex	
Female	
Male 🔤	
Male Female	
Male	
Male	
Male	-

↑ 3.6.10 List Samples

# CheckBox

# **Field Name**

ACTIVE (Alpha 1)

# Description

How to visualize a field as a Check Box in a grid.

### Uses

This Visualization type is useful for a Field that is used as a Boolean(TRUE\FALSE).

# Details

- 1. Through the Field Component source add a Picklist visualization ( PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add PicklistItems for male and Female to the Picklist. Save the field.
- 2. Create a grid component. From the fields tab add the ACTIVE field to the grid as a column. Set the following properties on the column.
  - ReadOnly = FALSE.
  - DisplayAppearance=CheckBox
  - EditAppearance=CheckBox
  - UsePicklist = TRUE.
  - ColumnAlign = Center.
- 3. Load the grid and the column can be edited as a checkbox.

# **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('Yes') Default(Tru-Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('No') Parent(#Pick

End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

 $End\_Com$ 

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Columnalign(Center) Disj

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

Endroutine

End\_Com

# Appearance

Active	
	*

#### Note:

Clicking the check box will set the corresponding value for the field from the

picklist items. Values are chosen from the following rule.

- Checkbox Unchecked = First picklist item value
- CheckBox Checked = Last PicklistItem value
- CheckBox grayed = Any picklistItem between first and last or an invalid value. (eg Changing the value of the field to a value that is not the first or last item in the picklist will result in the checkbox being grayed.)

↑ 3.6.10 List Samples

# Image

# **Field Name**

NATION (Alpha 3)

# Description

How to visualize a field as an Image set in a grid.

## Uses

This Visualization type is useful to provide a more visual means of selecting values from a list.

# Details

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add picklist items to represent nationalities and attach an image to each picklist item. Change the Appearance property of the PRIM\_EVPL to Image. Save the field and use the VisualPicklist class on a form.

- 1. Through the Field Component source add a Picklist visualization ( PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add picklist items to represent nationalities and attach an image to each picklist item. Save the field.
- 2. Create a grid component. From the fields tab add the NATION field to the grid as a column. Set the following properties on the column.
  - ReadOnly = FALSE.
  - DisplayAppearance=Image
  - EditAppearance=Image
  - UsePicklist = TRUE.
  - ColumnAlign = Center.
- 3. Load the grid and the column is displayed and edited as an image set.

# **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('USA') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Japan') Image(#V

Define\_Com Class(#PRIM\_PKIT) Name(#Item) Caption('Great Britain') Imag Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Australia') Image End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

 $End\_Com$ 

 $End\_Com$ 

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Columnalign(Center) Disj

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

Endroutine

End\_Com

# Appearance

	0.5
Nationality	
Nationality	
A MARKET AND A MAR	-

↑ 3.6.10 List Samples

# ImageAndText

# **Field Name**

NATION (Alpha 3)

# Description

How to visualize a field as an Image set in a grid.

## Uses

This Visualization type is useful to provide a more visual means of selecting values from a list.

# Details

Through the Field Component source add a Picklist visualization (PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add picklist items to represent nationalities and attach an image to each picklist item. Change the Appearance property of the PRIM\_EVPL to Image. Save the field and use the VisualPicklist class on a form.

- 1. Through the Field Component source add a Picklist visualization ( PRIM\_EVPL) and a Picklist. Note that these are separate classes in the Field source. Add picklist items to represent nationalities and attach an image to each picklist item. Save the field.
- 2. Create a grid component. From the fields tab add the NATION field to the grid as a column. Set the following properties on the column.
  - ReadOnly = FALSE.
  - DisplayAppearance=ImageAndText
  - EditAppearance=ImageAndText
  - UsePicklist = TRUE.
  - ColumnAlign = Center.
- 3. Load the grid and the column is displayed and edited as an image set.

# **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('USA') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Japan') Image(#V

Define\_Com Class(#PRIM\_PKIT) Name(#Item) Caption('Great Britain') Imag Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Australia') Image End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance(

End\_Com

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Columnalign(Center) Disj

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

Endroutine

End\_Com

#### Appearance

Nationality	
Japan	
Great Britain	
Australia	
Japan	
📥 Japan	
USA	*

# ↑ 3.6.10 List Samples

## **Reusable Part**

# **Field Name**

SECTION(Alpha 2)

# Description

How to visualize a field as an ReusablePart in a grid.

You can edit a field in a list as a reusable part. The first step to this is to create the reusable part. The part form MUST have a default property in order to work as expected.

# How to create the sample

- 1. Create a reusable part called SECTION. Paste the reusable part code from below
- 2. Create a form and copy the form code from below. Compile and run.

# How it works

When creating a reusable part we ensure we follow the reusable part guidelines. We add a property to the form as follows and ensure it is made the default property for the reusable part (eg Defaultpty(Section) )

Define\_Pty Name(Section) Get(Get\_Section) Set(Set\_Section)

The set of this property will be called when the user activates the cell in the grid. When called the property will be given the current value of the field. This is to allow the reusable part an opportunity to ensure that the correct section is selected prior to the display of the form.

The reusable part will then be made visible and the user is free to enter the new section. When the grid needs the new value for the cell it will call default property Get routine to retrieve the value and update the grid cell.

When time to commit the grid then calls the get of the default property to retrieve the reusable part value and updates its contents from the new value.

# **Reusable Part Source (SECTIONR)**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_PANL) Defaultpty(Section) Displaypor \* Reusable consists of a combo box with a client layout item attachment Define\_Com Class(#PRIM\_CMBX) Name(#SECT\_CBX) Displayposition(1) Define\_Com Class(#PRIM\_CBCL) Name(#CBCL\_1) Displayposition(1) Pare Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1) Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Center) Mana

Define\_Pty Name(Section) Get(Get\_Section) Set(Set\_Section)

Evtroutine Handling(#COM\_OWNER.Initialize) Options(\*NOCLEARMESS/

\* Select all entries from PSLMST and add to our combo box Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#SECT\_CBX)

Endselect

Endroutine

```
Ptyroutine Name(Get_Section)
Define_Map For(*output) Class(#SECTION) Name(#o_section)
```

\* Default property Get.

\*

\* Get\_Entry on the combo box focus item and set the return value

If\_Ref Com(#SECT\_CBX.focusitem) Is\_Not(\*NULL)

Change Field(#STD\_NUM) To('#SECT\_CBX.FOCUSITEM.ENTRY')

Get\_Entry Number(#STD\_NUM)

Endif

Set Com(#o\_section) Value(#SECTION)

Endroutine

Ptyroutine Name(Set\_Section) Define\_Map For(\*input) Class(#SECTION) Name(#i\_section) \* Default property Set.

\* Select\_List on Combo and set focus to mathing item from input

Selectlist Named(#SECT\_CBX)

If Cond('#SECTION \*ne #I\_SECTION')

Continue

Endif

Set Com(#SECT\_CBX.currentitem) Focus(true) Selected(true)

Endselect

Endroutine

End\_Com

#### **Form Source**

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(147) Clientwidth

Define\_Com Class(#PRIM\_GRID) Name(#GRID) Captionnoblanklines(True) Define\_Com Class(#PRIM\_GDCL) Name(#GDCL) Displayappearance(Edit) ]

Define\_Com Class(#SECTIONR) Name(#SECTIONR) Height(17) Left(8) Par

Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM

Select Fields(#SECTION) From\_File(PSLMST)

Add\_Entry To\_List(#GRID)

Endselect

Endroutine

End\_Com

# Appearance

Section Code	
DV	
03	
05	
03	
01	
02	
02	*

#### Note:

To allow for a section reusable part that is filtered by department you can add an event handler for the Grid.Changed event. This handler can then access another property on the reusable part that takes the department to filter by. The reusable part can then respond by updating its contents.

# 3.6.11 Miscellaneous

UsePicklist NoMatchAction ↑ 3.6 Field Visualization Development

## UsePicklist

# **Field Name**

STD\_NUM

# Description

Detail the behavior of the UsePicklist through the STD\_NUM field. PRIM\_EVEF, PRIM\_EVSE, PRIM\_GDCL, PRIM\_LVCL & PRIM\_TVCL all have a property called UsePicklist. This property is a design-time only property. The purpose of this property is to specify whether the picklist of a field, once specifed, is to be used as part of the formatting for the relevant component.

Example: A PRIM\_EVSE ( SpinEdit ) has been defined for a numeric field that also includes a Picklist. If the UsePicklist property is FALSE then when visualized the SpinEdit will work as normal through the range of allowable numbers. If UsePicklist=TRUE then the SpinEdit will cycle through the picklist values.

The primary purpose of this property is seen as a way to introduce Field Visualization concepts bit by bit to a large system. Older component can work as they always have with UsePicklist=FALSE whereas new components can have UsePicklist=TRUE to gain access to new functionality.

# Details

Through the field component source add the field source below. Save the field and then use the spinedit visualization on a form.

### **Field Source**

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Height(19) Usej

 $End\_Com$ 

Begin\_Com Role(\*Visual #PRIM\_EVSE) Name(#VisualSpinEdit) Defaultvisu

End\_Com

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('Item 1') Default('I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Item 2') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Item 3') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_4) Caption('Item 4') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_5) Caption('Item 5') Parent(#I

End\_Com

End\_Com

# Form Source ( Behavior with UsePicklist=FALSE )

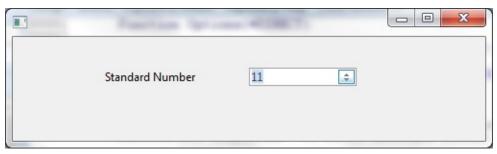
Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#STD\_NUM.Visual) Name(#STD\_NUM) Displayposition(

End\_Com

# Appearance ( Behavior with UsePicklist=FALSE )



Form Source ( Behavior with UsePicklist=TRUE )

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#STD\_NUM.Visual) Name(#STD\_NUM) Displayposition(

End\_Com

# Appearance ( Behavior with UsePicklist=TRUE )

Nationality	Item 4	

# ↑ 3.6.11 Miscellaneous

## **NoMatchAction**

# **Field Name**

STD\_NUM

# Description

Detail the behavior of the NoMatchAction property through the STD\_NUM field.

# Details

NoMatchAction is a property of the PRIM\_PKLT component and serves to allow developers to define the behavior for a picklist when a field is set to a value that is not part of its defined picklist. The below examples show the different behaviors for the values of this property.

# Field Source ( Behavior with NoMatchAction =Blanks )

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Height(19) Usej

End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVSE) Name(#VisualSpinEdit) Defaultvisu

 $End\_Com$ 

Begin\_Com Role(\*picklist) Name(#Picklist) Nomatchaction(Blanks)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('Item 1') Default('I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Item 2') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Item 3') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_4) Caption('Item 4') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_5) Caption('Item 5') Parent(#I

End\_Com

End\_Com

## Form Source ( Behavior with NoMatchAction =Blanks )

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#STD\_NUM.Visual) Name(#STD\_NUM) Displayposition( Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) Caption('Set Invalid Val

Evtroutine Handling(#PHBN\_1.Click)

\* As UsePicklist=TRUE the only allowable values for the STD\_NUM field are

\* The value 1234 is not in the Picklist for STD\_NUM so 1234 is an invalid val

\* then take presedence

Change Field(#STD\_NUM) To(1234)

Endroutine End\_Com

# **Appearance ( Behavior with NoMatchAction =Blanks )**

Standard Number	\$	
	Set Invalid Value	

**Field Source ( Behavior with NoMatchAction = ShowValue )** Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Height(19) Usej

End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVSE) Name(#VisualSpinEdit) Defaultvisu

 $End\_Com$ 

Begin\_Com Role(\*picklist) Name(#Picklist) Nomatchaction(ShowValue)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('Item 1') Default('I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Item 2') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Item 3') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_4) Caption('Item 4') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_5) Caption('Item 5') Parent(#I Define\_Com Class(#I Define\_COM Cla

End\_Com

End\_Com

# Form Source ( Behavior with NoMatchAction = ShowValue )

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#STD\_NUM.Visual) Name(#STD\_NUM) Displayposition( Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) Caption('Set Invalid Val

Evtroutine Handling(#PHBN\_1.Click)

\* As UsePicklist=TRUE the only allowable values for the STD\_NUM field are \* The value 1234 is not in the Picklist for STD\_NUM so 1234 is an invalid val \* then take presedence

Change Field(#STD\_NUM) To(1234)

Endroutine End\_Com

### Appearance ( Behavior with NoMatchAction = ShowValue )

Standard Number	1234 \$	
	Set Invalid Value	
	Sectivality volde	

**Field Source ( Behavior with NoMatchAction =DefaultItem )** Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Height(19) Usej

End\_Com

Begin\_Com Role(\*Visual #PRIM\_EVSE) Name(#VisualSpinEdit) Defaultvisu

 $End\_Com$ 

Begin\_Com Role(\*picklist) Name(#Picklist)

Define\_Com Class(#PRIM\_PKIT) Name(#Item\_1) Caption('Item 1') Default('I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_2) Caption('Item 2') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_3) Caption('Item 3') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_4) Caption('Item 4') Parent(#I Define\_Com Class(#PRIM\_PKIT) Name(#Item\_5) Caption('Item 5') Parent(#I

End\_Com

End\_Com

#### **Form Source ( Behavior with NoMatchAction = DefaultItem)** Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(108) Clientwidth

Define\_Com Class(#STD\_NUM.Visual) Name(#STD\_NUM) Displayposition( Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) Caption('Set Invalid Val Evtroutine Handling(#PHBN\_1.Click)

\* As UsePicklist=TRUE the only allowable values for the STD\_NUM field are

\* The value 1234 is not in the Picklist for STD\_NUM so 1234 is an invalid val

\* then take presedence

Change Field(#STD\_NUM) To(1234)

Endroutine End\_Com

# Appearance ( Behavior with NoMatchAction = DefaultItem)

Standard Number	Item 1	÷	
	Set Invali	d Value	

**Note:** The NoMatchAction property only has meaning when UsePicklist=TRUE.

↑ 3.6.11 Miscellaneous

# **3.6.12 Field Visualization in Depth**

This section describes in depth field visualization concepts. Fields as Components Field Class list Field variable values vs. Field Visualization appearance Field variable default value vs. Field Visualization Picklist default item Field Visualization Q & A  $\uparrow$  3.6 Field Visualization Development

# **Fields as Components**

Components in Visual LANSA are defined using a BEGIN\_COM/END\_COM pairing. Be they a Form, Reusable Part or other component type all code for the component is within this BEGIN\_COM/END\_COM pairing. For Field Visualization what we have done is exposed the component side of a field for end user modification. What this means is that you can open a Field in the component editor and modify its code as you would any other Visual LANSA component.

This does not mean that you can start creating events and using all Visual LANSA commands within the component aspects of a field (as most commands are illegal and will flag as an error) But it does mean that certain aspect of the Visual LANSA language have been exposed to help end user modification of the component aspects of a field.

To explain further the reason that all LANSA commands are not available from the Field component source is that the Field source is never executed. All code entered is used to only construct the component form of the Field. Therefore all LANSA commands that are execution related will flag an error in the Editor.

↑ 3.6.12 Field Visualization in Depth

# **Field Class list**

Fields contain multiple classes. Each class is a single element representing a feature of Field Visualization. It is through the creation of these classes that you will access the various features of Field Visualization. For example, to visualize a field as a SpinEdit on a Form then a SpinEdit class (PRIM\_EVSE) will be added to the Field Class List in the field component source.

# Visualization Classes

These are the built in visualizations provided by LANSA through existing and new PRIM\_'s. The following is the complete set of built in visualizations

- PRIM\_EVEF Edit box
- PRIM\_EVSE Spin edit
- PRIM\_EVPL Picklist
- PRIM\_EVTB Trackbar
- PRIM\_EVPB Progress Bar

Following is a small sample of the component code for a field that will be visualized as a SpinEdit:

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVSE) Name(#SPINEDIT) DefaultVisual(1

End\_Com

End\_Com

# **Reusable Part Visualization Classes**

Further to the built in Visualizations it is possible to Visualize your fields as a reusable part that you have created. This is a powerful feature that will be used when the built in types do not offer enough flexibility. Here is some sample component code for a field visualized as a reusable part.

```
Begin_Com Role(*EXTENDS #PRIM_OBJT)
```

Begin\_Com Role(\*VISUAL\_PART #PARTNAME) Name(#Custom) Defaultv

End\_Com

End\_Com

# **Prompter Classes**

Prompters are another new feature of Field Visualization. However, they are not visualizations themselves. They are form components that will be used to retrieve a value for the field when activated. The prompter is to be used in conjunction with the LANSA Visualization classes or the PRIM\_GRID (Grid), PRIM\_LTVW (ListView) and PRIM\_TRVW (Tree) controls. When visible the prompter can be accessed via the F4 key or by clicking on a button with the ellipses (eg ... ). Samples are included below but for here is a sample field component source including a prompter form called PROMPTER. This source will make a prompter available for the VisualEdit visualization.

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVEF) Name(#VisualEdit) Defaultvisual(T

End\_Com

Begin\_Com Role(\*prompter #AFLDVIS0P) Name(#PROMPTER)

End\_Com

End\_Com

# **Picklist Classes**

Picklists are another new feature of Field Visualization. Again, they are not visualizations themselves but are used in conjunction with other visualizations. A picklist is to be created when a field can contain only a predetermined set of values. One good example of this is a field called SEX. For this field it would be logical to create a picklist that contained 2 entries (one for Male and one for Female). These picklist values are then the only allowable visual values for that field.

Samples can be found below but once again here is the component source for a simple field called Sex that will be visualized as a picklist with a button set.

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#PICKLIST) Defaultvisual(T

End\_Com

Begin\_Com Role(\*picklist) Name(#LIST) Define\_Com Class(#PRIM\_PKIT) Name(#ITEM1) Caption('Male') Default(Tr Define\_Com Class(#PRIM\_PKIT) Name(#ITEM2) Caption('Female') Parent(# End\_Com

End\_Com

1 3.6.12 Field Visualization in Depth

# Field variable values vs. Field Visualization appearance

An important concept of Field Visualization is distinguishing the value of the field from its current visualization. It is possible for a field value to be set to a value that is not compatible with how that field is currently visualized. In this case the field variable and its visualization will be different.

Example. Define a field called SEX(Alpha(1)). Add a picklist and two picklist items to the field from its component source. Set captions and values for the two picklist items. Visualize the field as a ButtonSet on a form.

For the above scenario code can be compiled that allows the setting of the field to any value even though the Visualization for that field will only respond correctly to being changed to a value in the picklist.

The NoMatchAction property of the Picklist should be used in this case to define the behavior to be taken by the Visualization.

As the name suggests, this feature is related to the appearance of a field, and does not necessarily affect the value of the underlying variable at run time. The Visualization will attempt to show the value of the field where possible. However, it is possible to have a value in the underlying variable that is NOT in the PICKLIST.

↑ 3.6.12 Field Visualization in Depth

# Field variable default value vs. Field Visualization Picklist default item

Fields can have a default value and the picklist for a field can have a default item. If not careful it is possible for these two values to be out of sync with each other.

For example, If you run the below form in debug mode and look at the value of #EMPSEX at the "message" command it's value is blank unless the radio button has been selected. This may appear incorrect as the field picklist has a default item but is in fact correct behavior.

What happens is the form on start up initializes the field so #EMPSEX to its default value. In this case no default value has been specified so the field variable value is blank. However, for the visualization Blank is not a valid value so the visualization initialises as per the rules of the picklist (to the value of the default item in this case)

What we have created is a case where the field default value and the field picklist default items value are out of sync. To resolve this #EMPSEX should be defined with a default value of 'M'. Field and Visualization will then be in sync and the field value will match the visualized appearance

To avoid confusion it is recommended that a field default value be set to the value of the default item in the fields picklist.

#### Form Source

FUNCTION OPTIONS(\*DIRECT) BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) CLIENTHEIGHT(306) C DEFINE\_COM CLASS(#EMPSEX.Visual) NAME(#EMPSEX) DISPLAYPO DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#PHBN\_1) CAPTION('Save

EVTROUTINE handling(#com\_owner.Initialize) SET #com\_owner caption(\*component\_desc) ENDROUTINE

EVTROUTINE HANDLING(#PHBN\_1.Click) message 'Field Sex has a Value?' ENDROUTINE END\_COM

#### Field Source ( EMPSEX Alpha(1) )

#### BEGIN\_COM ROLE(\*EXTENDS #PRIM\_OBJT)

#### BEGIN\_COM ROLE(\*Visual #PRIM\_EVEF) NAME(#VisualEdit) HEIGHT(

END\_COM

\* Define Picklist Visualization BEGIN\_COM ROLE(\*Visual #PRIM\_EVPL) NAME(#PICKLIST) DEFAUL' End\_Com

\* Define the Picklist BEGIN\_COM ROLE(\*picklist) NAME(#LIST) DEFINE\_COM CLASS(#PRIM\_PKIT) NAME(#ITEM1) CAPTION('Male') I DEFINE\_COM CLASS(#PRIM\_PKIT) NAME(#ITEM2) CAPTION('Female' End\_Com

BEGIN\_COM ROLE(\*Visual #PRIM\_EVPL) NAME(#Dropdown) APPEAR

END\_COM END\_COM

# Field Visualization Q & A

# Q. Why is there no EditAppearance called DropDown or Combo?

A. In a list component an EditAppearance of combo box is handled by the existence of a picklist in the fields component source and the column property UsePicklist being set to true. This is why there is no DropDown or ComboBox EditAppearance.

## Q. Where is my ComboBox?

A. If you are trying to edit a field as a combo box in a grid and it is not displaying ensure you have taken the following steps.

Field being edited has a picklist defined in its component source Column ReadOnly property is set to FALSE Column UsePicklist property is set to TRUE

#### Q. I have created a picklist with a DefaultVisual(true) for a PRIM\_EVPL. This component also has an Appearance(DropDown) property. Why then is my field not a dropdown when used in a column in a grid?

A. The UsePicklist property for your column is set to FALSE. Setting this property to TRUE should make the DropDown available. If not then follow the sample above detailing how to display a dropdown in a grid.

# Q. Why does a picklist item text appear in a list when I do an Add\_Entry to a list?

A. If a picklist is being used for a field in a list then the value of the field at the time of the Add\_Entry and the NoMatchAction property of the picklist control what will be displayed to the user. Refer to the NoMatchAction example for further information.

### Q. What is the UsePicklist property?

A. PRIM\_EVEF, PRIM\_EVSE, PRIM\_GDCL, PRIM\_LVCL & PRIM\_TVCL all contain a property called UsePicklist. This property is a designtime only property. The purpose of this property is to specify whether the picklist of a field, once specifed, is to be used as part of the formatting for the relevant component.

For example: A PRIM\_EVSE ( SpinEdit ) has been defined for a numeric field that also includes a Picklist. If the UsePicklist property is FALSE then when visualized the SpinEdit will work as normal through the range of allowable

numbers. If UsePicklist=TRUE then the SpinEdit will cycle through the picklist values.

The primary purpose of this property is seen as a way to introduce Field Visualization concepts bit by bit to a large system. Older component can work as they always have with UsePicklist=FALSE whereas new components can have UsePicklist=TRUE to gain access to new functionality.

## Q. What is the NoMatchAction property?

A. The NoMatchAction property is a property of PRIM\_PKLT and is used by picklist to specify the behavior when a field is set to a value that is not present in the picklist.

# **Q. What does default value for DisplayAppearance and EditAppearance mean?**

A. A value of Default in either the DisplayAppearance or the EditAppearance property signifies that LANSA is to query the current field characteristics and make a 'best guess' of the type of visualization that will be used for the field. This allows the component Editor to immediately provide visual feedback in the List components that support Field Visualization.

For an example take the following steps.

- 1. Create a new field (alpha 5)
- 2. Open the component source for the Field and add a Visual Picklist (PRIM\_EVPL)
- 3. Change the Appearance property of the VisualPicklist to CheckBox.
- 4. Change the DefaultVisual property of the VisualPicklist to TRUE.
- 5. Save the field and use it as a column in a grid.

The above steps show that a value of Default for DisplayAppearance and EditAppearance show items in the list as a check box.

Refer to feature help of these properties for further information on how LANSA interprets the field to find a suitable default.

The value 'Default' for DisplayAppearance and EditAppearance is just that, a Default. It is not possible for LANSA to correctly interpret the field for the desired appearance all the time so if the default value is not satisfactory the value of this property can always be changed.

# Q. Why can't I type into my EditBox Visualization?

A. If a field is visualized as an EditBox and you cannot type in the EditBox then

the UsePicklist property of the PRIM\_EVEF is probably set to TRUE. Switch it to FALSE and you should be able to Edit again.

# **Q.** What is the DefaultVisual property?

A. The DefaultVisual property specifies the default visualization for a field. Whichever visualization is specified as DefaultVisual(TRUE) will be the visualization used when referred to as FieldName.Visual

## Q. Why is it so hard?

A. Its not. If it seems to be then take a step backwards. Ensure you are familiar with all relevant terminology. Figure out what you are trying to accomplish first and then find the sample that is the closest match to your requirements.

# **Q.** The field name is VisualEdit but it is referred to as .Visual in the form source. Why?

A. The field can be used on a form as either .VisualEdit or .Visual. This is because the Field is sourced with the DefaultVisual(TRUE) property. Whichever class is defined with DefaultVisual(TRUE) can be referenced by its name or the .Visual identifier. This serves to allow users to change what type of class they want "across the board" by changing the class that has DefaultVisual(TRUE)

## Q. How do I pass values from my prompter to the calling field?

A. Define a property for your prompter form and make it the default property of the form. This property will then be used to transfers values from the field to you prompter form and back again.

# Q. Why can't I see my prompter?

If a prompter has been specified for a field and the ellipses are not available to make the prompter visible then the ShowPrompter property is probably set to FALSE. Set it to TRUE and the ellipses should be become available so the prompter can be activated.

#### Q. Are SpinEdit Visualizations only for use with numeric fields?A. No. SpinEdits can be used also for fields that have a picklist defined. In these cases the SpinEdit will cycle through the available items in the picklist.Q. Why is my Check Box appearing as greyed?

A. A field visualized as a Check Box will appear as Greyed when the value of the field is equal to any item between the first and last items in the picklist or a value that is not in the picklist.

# ↑ 3.6.12 Field Visualization in Depth

# **3.6.13 Dynamic Visualizations**

You can set up dynamic visualizations for fields. For example, you can add logic to autofill the value of a field, or you can specify that the values of a picklist are to be retrieved from a file at run time. That is, picklists can be created 'on the fly' rather than being defined as static lists. For details refer to:

Visual Hosts and Dynamic Picklists Interfaces and monitors Prim\_LM - Monitor Prim\_DC.iMonitorSubject Prim\_DC.iDynamicPicklist Prim\_DC.iContextualSubject Examples of using Visualization Interfaces and Monitors

# **Visual Hosts and Dynamic Picklists**

A Visual Host allows the user to define a reusable part to act as the edit portion of a field. This allows the user all of the flexibility of a reusable part while still taking advantage of LANSA Repository features for multilingual descriptions, labels and column headings.

Dynamic Picklists allow a user to define the items in a picklist at runtime by loading them from a file for example.

1 3.6.13 Dynamic Visualizations

# **Interfaces and monitors**

Three new interfaces have been defined to support the new visualizations. An interface can be thought of as a contract between two objects. The object that implements the interface "promises" to provide the functionality as specified in the interface definition. By doing so, the object communicating with it can see that the interface has been implemented and thus "knows" that certain methods can be called. LANSA does not check to ensure that methods are redefined by the implementer. An error will occur at runtime if the required methods are called but have not been redefined.

Unlike inheritance, a component can implement more than interface, allowing it to support multiple behaviors depending on requirements.

To use an interface, add \*Implements and the name of the interfaces as shown following:

Begin\_Com Role(\*Extends #Prim\_panl \*implements #Prim\_dc.iMonitorSubje

### 1 3.6.13 Dynamic Visualizations

## **Prim\_LM** - Monitor

A monitor has a Source and a Target and allows any component that implements iMonitorSubject to communicate automatically with each other. By using a monitor, field dependencies and hierarchical dependencies can be easily managed without the need for complex coding.

### Prim\_DC.iMonitorSubject

**iMonitorSubject** is the simplest and most common form of Visual Host. All visual field instances implement this interface. It has two methods that will be called depending on how the field is used.

**ApplyMonitoredValue** is typically called when the field is used as the target of a monitor. A reference to the Monitor Source component is received in the MonitorSubject input map.

**GetValue** can be called to obtain the value of the field. As GetValue applies to any field type, the result map is a variant.

The **ValueChanged** event that can be signaled to indicate that the value of the variable has changed. This equates to the Changed event.

# Prim\_DC.iDynamicPicklist

Defines the interface you need to implement when you create a Dynamic Picklist visualization.

Load – depending on the properties of the Visual Picklist, this method will get called either once or many times to allow you to (re-)populate your Picklist using RDMLX.

# Prim\_DC.iContextualSubject

Implementing iContextualSubject allows a visualization reusable to be used for more than one field. Typically, visualizations are specific to the field they represent, but in certain circumstances reusing the same reusable part for multiple fields is of great value e.g. Code tables.

OnContextChanged – OnContextChanged will be executed when the visualization is first initialized or when the underlying dataclass changes e.g. Different columns in a grid. It is also called when certain field features are modified e.g. when the Readonly and ShowError properties are changed. A reference to the instance using the visualization is received in the Context input map providing access to the features of the container.

### **Examples of using Visualization Interfaces and Monitors**

Example 1 - Simple Hosted Part

Example 2 - Simple Dynamic Picklist

Example 3 – Linking Multiple Visualizations

#### **Example 1 - Simple Hosted Part**

Create a reusable part called FLDVIS01 using the source following. This is will provide autocompletion functionality for field DEPTMENT:

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_PANL \*implements #Prim\_dc.iMonito

Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1) Define\_Com Class(#DEPTMENT.VisualEdit) Name(#Department) Displaypos Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Center) Mana

Define\_Pty Name(Value) Get(GetPropertyValue) Set(SetPropertyValue)

Ptyroutine Name(GetPropertyValue) Define\_Map For(\*Output) Class(#prim\_alph) Name(#Property)

#Property := #Department

Endroutine

Ptyroutine Name(SetPropertyValue) Define\_Map For(\*Input) Class(#prim\_alph) Name(#Property)

#Department := #Property

Endroutine

Mthroutine Name(ApplymonitoredValue) Options(\*redefine)

\* No redefinition required

Endroutine

Mthroutine Name(GetValue) Options(\*redefine)

\* No redefinition required

Endroutine

Evtroutine Handling(#Department.KeyPress) Handled(#Handled) Keycode(#K

```
* If the field isn't full
If (#Department.CurSize <> #Department.FieldLength)
```

```
* If a character entered
If (#KeyCode = isChar)
```

#Com\_Owner.PrepareAutoComplete

If (#Com\_owner.CanAutoComplete)

#Handled := True

#Com\_owner.AutoComplete( #Char )

Signal Event(ValueChanged)

Endif

Endif

Endif

Endroutine

Evtroutine Handling(#Department.Changed)

\* Handle all other key presses that might affect the value Signal Event(ValueChanged)

Endroutine

Mthroutine Name(CanAutoComplete) Help('Can we autocomplete?') Access(\* Define\_Map For(\*Result) Class(#prim\_boln) Name(#Result)

\* If selection doesn't start at the end of the value, autocomplete is not appropria #Result := (#Department.SelectionEnd = (#Department.Trim.cursize + 1)) Endroutine

Mthroutine Name(AutoComplete) Access(\*private) Define\_Map For(\*Input) Class(#prim\_alph) Name(#Char) Help('Character just

Define\_Com Class(#prim\_nmbr) Name(#Start)
Define\_Com Class(#prim\_alph) Name(#Candidate)

#Start := #Department.SelectionStart

#Candidate := #Com\_owner.PrepareCandidate( #Char )

#Department := #Com\_owner.GetCandidate( #Candidate )

```
* Set selection to be startposition + 1 to the end
#Department.SelectionStart := #Start + 1
#Department.SelectionEnd := #Department.Trim.cursize + 1
```

Endroutine

Mthroutine Name(PrepareAutoComplete) Help('Prepare Selection in the value

Define\_Com Class(#prim\_nmbr) Name(#Transition)

\* If Start is greater than end, reverse the selection points If (#Department.SelectionStart > #Department.SelectionEnd)

#Transition := #Department.SelectionStart

#Department.SelectionStart := #Department.SelectionEnd

#Department.SelectionEnd := #Transition

Endif

Endroutine

Mthroutine Name(PrepareCandidate) Help('Prepare the input value ready for lc

Define\_Map For(\*Input) Class(#prim\_alph) Name(#Char) Help('Character just Define\_Map For(\*Result) Class(#Prim\_alph) Name(#Result)

\* If selection is the whole word, only use the char supplied by the event If (#Department.SelectionStart = 1)

#Result := #Char.uppercase

Else

\* Get anything to the left of the cursor start position and append the last key pr #Result := (#Department.substring( 1 (#Department.SelectionStart - 1) ).trim +

Endif

Endroutine

```
Mthroutine Name(GetCandidate) Access(*private)
Define_Map For(*Input) Class(#prim_alph) Name(#Candidate)
Define_Map For(*Result) Class(#prim_alph) Name(#Result)
```

\* If no record found, the last value entered is still the right answer #Result := #Candidate

\* Find the first record starting with the candidate value Select Fields(#Deptment) From\_File(Deptab) With\_Key(#Candidate) Generic(

```
#Result := #Deptment
```

Leave

Endselect

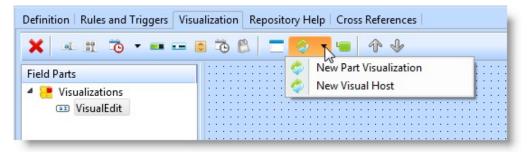
Endroutine

End\_Com

Note how the reusable part only displays a single input box for DEPTMENT.

No labels or descriptions are required.

Open field DEPTMENT and turn to the visualization tab.



Select the option for New Visual Host and the choose FLDVIS01 from the prompt. DEPTMENT should now appear as follows:

💥 💷 🗄 🗉 🕾 🚍 🖺 📋 🤌 🕶 🌴 🦫						
Field Parts						
Visualizations	Department Code	ABCD				
च्च VisualEdit च्य VisualHost	Department Code					
Source * Begin_Com Role(*EXTENDS	#PRIM_OBJT)					
Begin_Com Role(*Visual #PRIM_EVEF) Name(#VisualEdit) Defaultvisual(True) Height(19) Usepicklist(False) Width(201)						
End_Com						
	Host #FLDVIS01) Name(#Vi	isualHost) Height(19) Width(209)				
End_Com						
End_Com		and a second				

The new Visual\_host reusable can now be used as follows:

Define\_Com Class(#Deptment.Visualhost) Name(#...)

You might also consider changing the default name of VisualHost to AutoComplete or similar. This means when it's used on a form you'll see the following:

Define\_Com Class(#Deptment.AutoComplete) Name(#...)

#### **Example 2 - Simple Dynamic Picklist**

Create a reusable part called FLDVIS02 using the source code following. This is will provide picklist contents for DEPTMENT:

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_OBJT \*implements #Prim\_dc.iDynam

Mthroutine Name(Load) Options(\*redefine)

#Picklist.RemoveAll
Select Fields(\*all) From\_File(Deptab)
#Picklist.Add( #Deptment #Deptdesc )
Endselect

Endroutine

End\_Com

Ald Parts Visualizat Visualedit Visualedit VisualHost	
End_Com	al #PRIM_EVEF) Name(#VisualEdit) Defaultvisual(True) Height(19) Usepicklist(False) Width(201)
Begin_Com Role("Visu	al_Host #FLDVIS01) Name(#VisualHost) Height(19) Width(209)

Use the option to add a dynamic picklist. From the prompter, select FLDVIS02. The following code will be added to the visualization source code:

Begin\_Com Role(\*Dynamic\_Picklist #FLDVIS02) Name(#DynamicPicklist) End\_Com Begin\_Com Role(\*Visual #PRIM\_EVPL) Name(#VisualPicklist) Appearance( End\_Com

As with all picklists, dynamic or static, the NoMatchAction property needs to be set to determine what in the event of the underlying field value not being found in the picklist. For most situations, ShowValue is probably the best choice.

The new Visual\_host reusable can now be used as follows:

Define\_Com Class(#Deptment.VisualPicklist) Name(#...)

### **Example 3 – Linking Multiple Visualizations**

Having a created a dynamic picklist for Department, it is not unreasonable that a picklist might also be required for Section and subsequently employee.

The code following can be used for Section. As with the previous example, the Load method populates the picklist. However, as Section needs to be linked to Deptment at run time the reusable part also implements iMonitorSubject.

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_OBJT *implements #Prim_dc.iDynam
```

Mthroutine Name(Load) Options(\*redefine)

#Picklist.RemoveAll

```
Select Fields(*all) From_File(Sectab) With_Key(#Deptment)
```

```
#Picklist.Add( #Section #Secdesc )
```

Endselect

Endroutine

```
Mthroutine Name(ApplyMonitoredValue) Options(*Redefine)
```

```
#Deptment := #MonitorSubject.GetValue
```

Endroutine

End\_Com

The form following makes use of both the Deptment and Section visualizations. A monitor is defined with a source of Deptment and a target of Section. When Deptment changes, the ApplyMonitorValue method in the Section visualization is run. A reference to the source object is received allowing the GetValue method to be called to obtain the value.

Immediately after the ApplyMonitorValue has finished, the Load method will be run.

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(304) Clientwidth

Define\_Com Class(#Deptment.VisualPicklist) Name(#Deptment) Displayposit Define\_Com Class(#Section.VisualPicklist) Name(#Section) Displayposition(2

Define\_Com Class(#prim\_lm) Name(#DepartmentSection) Source(#Deptment

Evtroutine Handling(#Deptment.Changed)

\* Ensure the Section is valid after a department change #Section := \*null

Select Fields(#Section) From\_File(sectab) With\_Key(#Deptment) Leave Endselect

Endroutine End\_Com

Note how it is still necessary to ensure that the variable values in the form are maintained. When Deptment changes the user must ensure that section is changed to a reasonable value, in this case, the first section in the table for the new department value.

The code following is for a dynamic picklist for Empno.

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_OBJT *implements #Prim_dc.iDynam
```

Mthroutine Name(Load) Options(\*redefine)

#Picklist.RemoveAll

Select Fields(\*all) From\_File(pslmst1) With\_Key(#Deptment #Section)

#Picklist.Add( #Empno ("&1 &2 (&3)").Substitute( #Givename #Er

Endselect

Endroutine

```
Mthroutine Name(ApplyMonitoredValue) Options(*Redefine)
```

Case ((#MonitorSubject \*As #Prim\_objt).ComponentPatternName)

```
When (= Deptment)
#Deptment := #MonitorSubject.GetValue
```

```
When (= Section)
#Section := #MonitorSubject.GetValue
```

Endcase

Endroutine

End\_Com

This is much the same, in concept, as the code for section. However, note the change to ApplyMonitorValue. Empno needs to know the value of both Deptment and Section. As the iMonitorSubject received is a reference to the source component you can use the ComponentPatternName to determine which variable has been changed.

If Empno is then added to the form, as following, a further two monitors are required to inform Empno of changes to Deptment and Section.

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(304) Clientwidth

Define\_Com Class(#Deptment.VisualPicklist) Name(#Deptment) Displayposit Define\_Com Class(#Section.VisualPicklist) Name(#Section) Displayposition(2 Define\_Com Class(#EMPNO.VisualPicklist) Name(#Empno) Displayposition(

Define\_Com Class(#prim\_lm) Name(#DepartmentSection) Source(#Deptment Define\_Com Class(#prim\_lm) Name(#SectionEmployee) Source(#Section) Ta Define\_Com Class(#prim\_lm) Name(#DepartmentEmployee) Source(#Deptme

Evtroutine Handling(#Deptment.Changed)

\* Ensure the Section and Empno are valid after a department change #Com\_owner.GetDefaultSection #Com\_owner.GetDefaultEmpno Endroutine

Evtroutine Handling(#Section.Changed) \* Ensure the Empno is valid after a department change #Com\_owner.GetDefaultEmpno Endroutine

Mthroutine Name(GetDefaultSection) Access(\*private)

#Section := \*null

Select Fields(#Section) From\_File(sectab) With\_Key(#Deptment) Leave Endselect

Endroutine

Mthroutine Name(GetDefaultEmpno) Access(\*private)

```
#Empno := *null
```

Select Fields(#Empno) From\_File(pslmst1) With\_Key(#Deptment #Section) Leave Endselect

Endroutine

 $End\_Com$ 

# 4. Creating Application Databases

Before creating new files for your application database, review the following topics:

- 4.1 Getting Started with Database
- 4.2 Developing Databases
- 4.3 Logical View Development
- 4.4 Virtual Field Development
- 4.5 File Rules & Triggers Development
- 4.6 Access Route Development
- 4.7 Predetermined Join Field Development
- 4.8 Batch Control Development
- 4.9 Compiling Files
- 4.10 Load Other File

#### Also See

Files in the *Technical Reference Guide* Editing Files in the *User Guide* 

## 4.1 Getting Started with Database

Before starting to create new files in the repository, review the following topics:

- 4.1.1 Modeling Databases
- 4.1.2 File Standards and Guidelines
- 4.1.3 Before You Begin Checklist

#### Also See

- LANSA Application Design Guide
- 1 4. Creating Application Databases

## 4.1.1 Modeling Databases

A model is a representation or a simplified copy of something more complex. You use models to examine large or complicated structures which are difficult to work with directly. For example, models of air plane wings are tested and scale models of buildings are created. A data model is a tool which allows the developer to construct a picture of the data within an organization or a part of an organization. It is not a picture of what the computer sees, but of what the user and business sees.

Logical Data Models are high level or conceptual models which describe data in terms of the user perspective and business rules. A logical data model represents the business. A logical data model looks at the basic pieces of information and the inter-dependencies between the information. Logical data models are defined in terms of entities, attributes, and relationships. They reflect the business, its information and rules. It is not designed to represent the Architect's perception of the physical database to be implemented.

LANSA provides a modeling tool that allow the systems analyst to build up and manipulate a complete logical data model comprising elements, entities and relationships and applies data analysis rules to ensure that the model is properly normalized. This tool allows the physical database to be generated from the logical model, but recognizes the fact that there will be differences between the two. For example, for performance reasons, you may not wish to implement a fully normalized database.

A modeling tool documents the logical data model in such a way that it is understood by users. It can be reviewed and verified by the users as part of the analysis/design process before any physical database is built.

A modeling tool improves the developers productivity by automating the build of the physical database. The modeler helps the developer to create quality databases with a consistent standard. It supports the concept of a corporate data types and allows the developer to define standards for the implemented physical database.

If you are building a new application or re-engineering an application, it is highly recommended that you use a modeling tool to build your application database. (Refer to Using a Modeling Tool.)

1 4.1 Getting Started with Database

## 4.1.2 File Standards and Guidelines

Standards are needed to ensure consistency in your application database and to protect your investment in the repository. The following file naming recommendations are provided for "in house" application development, (as opposed to vendor "package" development).

File names should be in the format xxxxxnn where:

- A physical file name should be 6 characters or less and should not contain an '#' or '@' symbol.
- The xxxxx part of the name should be 1 to 6 characters and as meaningful as possible given the size limitation. For example: CUSTMR- Customer Master, SALEHS- Sales History.
- File names should be no longer than eight characters.
- Logical file names should use the base file name xxxxx and add a sequence number nn . For example: CUSTMR01, CUSTMR02, CUSTMR03.
- 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.

Please review the LANSA Object Names in the LANSA Technical Reference *Guide* 

#### Also See

LANSA Application Design Guide

1 4.1 Getting Started with Database

# 4.1.3 Before You Begin Checklist

Before you begin you development of files in the Repository, review the following checklist:

- The purpose, concepts and architecture of the LANSA Repository are understood so that your development efforts will properly centralize application information.
- All repository development procedures, standards and guidelines are known by/available to every member of the project team. For information, refer to Controlling Repository Maintenance and Development Environment Administration.
- LANSA Object naming standards and guidelines are known by/available to every member of the project team. For information, refer to 4.1.2 File Standards and Guidelines and the LANSA Object Names in the LANSA Technical Reference Guide.
- All file level repository features have been identified and development impacts are understood. Refer to 4.2.5 Steps for Completing a Database in LANSA.
- You have read 4.2 Developing Databases and you have identified the methods that will be used to populate the repository.

#### Also See

LANSA Application Design Guide

1 4.1 Getting Started with Database

## 4.2 Developing Databases

As you begin to create new files in the Repository, you should be familiar with the following topics:

- 4.2.1 LANSA File Concepts
- 4.2.2 Types of Fields in a File Definition
- 4.2.3 How to Create Files in LANSA
- 4.2.4 Adding Real Fields to a File
- 4.2.5 Steps for Completing a Database in LANSA
- 4.2.6 File Definition Attributes

#### Also See

Files in the *Technical Reference Guide*Editing Files in the *User Guide*↑ 4. Creating Application Databases

# 4.2.1 LANSA File Concepts

A file is a database table in which records can be retrieved, added, changed or deleted. In IBM i terminology, a file may be a physical file or a logical file. Logical files are also known as views in some database terminology.

To create a file, LANSA uses a Repository-based File Definition. This file definition is a database independent definition of the file characteristics.

A file definition contains:

- The real field in the file that make up the format of the file records. Some keys may be nominated as keys to the file.
- **4.3 Logical View Development** of the file that are required. The logical views may contain select/omit criteria.

In addition the LANSA file definition may also contain:

- 4.4 Virtual Field Development that are fields which are not actually present in the database file but are derived from fields that are in it. Hence the name virtual. These fields appear as if they were in the file, even though they are not.
- 4.5 File Rules & Triggers Development that are to be performed on fields when records are added, updated or deleted.
- **4.6** Access Route Development from the current file to other files in the system. This facility allows users to access a complete database without having a detailed knowledge of its structure.
- 4.7 Predetermined Join Field Development which use the current file's access routes to retrieve information from related files.
- **4.8** Batch Control Development which specifies other files in the system that contain batch, control or summary totals of information in the current file. LANSA will ensure that the summary or control information in other files is automatically maintained.

Some points to note about working with file definitions are:

- The file definition name and actual physical file name are always the same. When LANSA requests that the name of a file definition be supplied, specify the name of the physical file (as opposed to the name of any associated logical views/ files).
- The allowed field types in a file are determined by the partition settings. For more details, refer to RDML and RDMLX Partition Concepts in the

Administrator Guide.

- Within the LANSA System, logical views/files are considered to be part of the file definition.
- The file definition contains the attributes of the associated physical file and all associated logical views/files.

↑ 4.2 Developing Databases

# 4.2.2 Types of Fields in a File Definition

LANSA file definitions can contain 3 types of fields:

#### **Real fields**

These are the normal fields found in any file definition and are assembled together to form the record format of the associated physical file. Such fields actually exist in the database file and their content (or value) can be extracted from any record in the file at any time. (Refer to 4.2.4 Adding Real Fields to a File.)

#### Virtual fields

These are fields that do not actually exist in the IBM i database file, but are dynamically derived from "real" fields in the file. (Refer to 4.4 Virtual Field Development.)

#### **Predetermined join fields**

These are fields defined on access routes to receive information retrieved from the file accessed by the access routes. (Refer to 4.6 Access Route Development and 4.7 Predetermined Join Field Development.)

1 4.2 Developing Databases

## 4.2.3 How to Create Files in LANSA

Files can be created several ways in LANSA:

- 1. Manually define a file
- 2. Copy an existing file in the Repository
- 3. Load an existing externally defined file (OTHER file) into the Repository
- 4. Use a modeling tool.

**Option 1**. You may manually create all the details of the file definition. If you are not using the logical modeler, then all your fields will be created manually.

**Option 2**. You may create a new file by copying an existing file definition in the repository. LANSA will copy all of the file definitions details (rules, trigger, logicals, etc.).

**Option 3**. Loading Existing File Definitions. If you are building from existing applications, you can load your existing file or table definitions into LANSA. (Refer to 4.10 Load Other File.)

**Option 4**. There are several different options for data modeling with LANSA. The Visual LANSA Logical Modeler can be used to create your database. If using LANSA for i, the LANSA for i Modeler can be used to build your file definitions for a new application. (Refer to Using Modeling Tools.)

#### Also See

Creating Files in the User Guide

1 4.2 Developing Databases

## 4.2.4 Adding Real Fields to a File

If you think of a file as a table, the fields in the file are the columns in the table that make up the record format. To complete a basic file definition, you must identify the following:

- What are the fields in the file?
- What is the sequence of these fields?
- What fields are used as the keys to the file?

There are two ways of adding the real fields to a file.

### Add Single Field

You may add an existing field in the repository to a file definition. When the field is added, you can specify if it is a key field and where it is sequenced in the file.

#### Add Multiple Fields

You may multiple fields to a file definition. Using this approach, the field does not have to exist in the repository. You can specify the field characteristics as you add them to the file and they will be created in the repository once you have completed the list of fields. Again, you can specify if a field is a key to the file. Sequence is based on the order that the fields are added.

#### Also See

Real Fields in File in the *Technical Reference Guide* ↑ 4.2 Developing Databases

#### 4.2.5 Steps for Completing a Database in LANSA

The basic development steps to manually build your LANSA application database are as follows:

- 1. Create a file definition.
- 2. Add the real fields to the file and identify key fields. Refer to 4.2.4 Adding Real Fields to a File.
- 3. Create logical views. Refer to 4.3 Logical View Development.
- 4. Add virtual fields to the file. Refer to 4.4 Virtual Field Development.
- 5. Add validation rules and triggers. Refer to 4.5 File Rules & Triggers Development.
- 6. Create access routes to other files in the database. Refer to 4.6 Access Route Development.
- 7. Add Predetermined Joined Fields. Refer to 4.7 Predetermined Join Field Development.
- 8. Make file operational. Refer to 4.9 Compiling Files.

If you have modeled a database or loaded other files to define a group of file, you will complete the following steps to complete the database:

- 1. Create additional logical views if required. Refer to 4.3 Logical View Development.
- 2. Add virtual fields. Refer to 4.4 Virtual Field Development.
- 3. Add validation rules and triggers. Refer to 4.5 File Rules & Triggers Development.
- 4. Create access routes to other files in the database. Refer to 4.6 Access Route Development.
- 5. Add Predetermined Joined Fields. Refer to 4.7 Predetermined Join Field Development.
- 6. Make files operational. Refer to 4.9 Compiling Files.

If you are deploying your database to another platform, you should review the 4.2.6 File Definition Attributes.

1 4.2 Developing Databases

# 4.2.6 File Definition Attributes

File definition attributes become increasingly more important as begin to deploy your applications to different databases,

Specialized database characteristics are supported by using the file definition attributes.

For example, High Speed Tables can provide very fast response time when used in an IBM i database. Specific files in your database may be well suited to use this specialized database characteristic on IBM i.

#### Also See

File Attributes in the Technical Reference Guide

1 4.2 Developing Databases

## 4.3 Logical View Development

Logical files or views are used to create alternate ways of organizing or accessing the data in your application database files. You can create different file sorting sequences that differ from the keys used in the physical file. To understand how to use logical views in LANSA, review the following topics:

- 4.3.1 Logical View Concepts
- 4.3.2 How to Create Logical Files
- 4.3.3 Logical File Considerations
- 4.3.4 Examples of Logical Files
- 4.3.5 Select/Omit Concepts

#### Also See

Logical Views in the *Technical Reference Guide* Logical Views Tab in the *User Guide* 

# 4.3.1 Logical View Concepts

Logical files or views are used to create alternate ways of organizing the data in your files. Logical files do not store data. For example, your customer physical file may be keyed or indexed by customer number but you need to be able to look up data by customer name. You could create a logical file which would put the customer file data in order by name. If you delete this logical file, the data is not deleted. The data is stored in the physical file or table.

Logical file definitions are created as part of the file definition process; i.e. a physical file must be defined before a logical file can be created over it. You create logical files by working with file definitions; however, with logical files you must work within a selected physical file definition. A single physical file may have many logical views of the data it contains.

Logical file definitions in LANSA include a file name and key fields. (Access path maintenance options are also specified for IBM i databases). In addition, logical files may have select and omit criteria specified which are used to control which records will be included in the logical view.

It is important to understand the basic concepts of databases and file definitions before beginning to work with LANSA. For more information, refer to the *IBM Programming Database Guide* or other appropriate database guides.

Remember, logical files are part of a LANSA file definition and are not separate objects in the repository. To create or edit a logical file, you must select and open the physical file that it is based upon.

#### Also See

- 4.3.4 Examples of Logical Files
- 4.3.3 Logical File Considerations
- 1 4.3 Logical View Development

## **4.3.2 How to Create Logical Files**

Logical files are part of a LANSA file definition. A logical file can only be created once the physical file has been defined to LANSA.

Logical files can be created several ways in LANSA:

- 1. Manually define a logical file
- 2. Load an existing externally defined file into the Repository
- 3. Use a modeling tool.

**Option 1**. If you manually define the files in the database, you must manually create the logical files for the database.

**Option 2**. If you load OTHER files, you are given the choice of which logical files you want to load into LANSA. (Refer to 4.10 Load Other File.)

**Option 3**. If you use the Visual LANSA Logical Modeler or the LANSA for i Modeler, many of the required logical files will be automatically created for your files based on the object relationships defined in the model and on any specified user views. (Refer to Using Modeling Tools.)

Once a new logical file have been added, you must 4.9.1 Making a File Operational.

#### Also See

Logical Views in the Technical Reference Guide

1 4.3 Logical View Development

## 4.3.3 Logical File Considerations

When planning for the use of logical files in your application database, there are a number of important considerations to make.

- Always be sure that the database architect fully understands performance issues of the DBMS which will be deployed when creating logical files. For example, on the IBM i, the use of immediate access path maintenance has performance implications when an IPL is performed or when saving objects.
- Also on the IBM i, LANSA supports the use of OPNQRYF. OPNQRYF provides extremely powerful and flexible methods of creating dynamic views over files.
- In general, it is a good practice to limit the number of logical views you create. Logical views should reflect frequently used access techniques for the files.
- RDML has many powerful file access techniques which can reduce the need for logical files. Be sure to review the capabilities of the RDML commands as part of your database design and requirements assessment.
- For RDMLX Files, the use of Select/Omit Criteria is not recommended. Instead, use the WHERE parameter to limit the data retrieved.

# 4.3.4 Examples of Logical Files

These examples apply to creating a logical file or view within a file definition. The file definition used in these examples is called CUSMST (Customer master file) and contains these fields:

Field	Description	Туре	Len	Dec
CUSTNO	Customer number (primary key)	S	7	0
NAME	Name of person / organization	А	30	
ADDR1	Address line 1	А	35	
ADDR2	Address line 2	А	35	
ADDR3	Address line 3	А	35	
POSTCD	Post code	А	4	
STATE	State mnemonic	А	3	
ACTIVE	Active / inactive flag	А	1	
CREDIT	Credit limit	Р	11	2
AMTDUE	Current amount due all accounts	Р	11	2
ACCTYP	Customer account type	Р	1	0

#### Example 1

Create a logical view/file called CUSMSTV1 that will order CUSMST by NAME and allow generic searching by NAME:

Logical view name : CUSMSTV1 Desc of logical view : Customer master by customer name Access path maint opt : IMMED Unique? NO Dynamic select? NO Key field details Field Description A/D S/U/A NAME\_\_\_\_\_\_A\_\_\_+ \_\_\_\_\_\_\_+

#### Example 2

Create a logical view/file called CUSMSTV2 that will order CUSMST by STATE then POSTCD (i.e. POSTCD within STATE):

Note that no ascend/descend value has been specified. In this case the value will default to ascend for both keys.

#### Example 3

Create a logical view/file called CUSMSTS1 that will order CUSMST by CUSTNO. Only records with ACTIVE = 'Y' are to be "visible" when using this logical view:

Example 4

Create a logical view/file called CUSMSTS2 that will order CUSMST by AMTDUE. Only records with CREDIT greater than 100000 and AMTDUE greater than or equal to 20000 are to be visible via this view. Records are to be

ordered from highest amount due to lowest amount due. Use the dynamic select feature as well.

Logical view name : CUSMSTS2

Desc of logical view : High credit, large debt customers Access path maint opt: IMMED Unique? NO Dynamic select? YES Key field details:

Field	Description	Α	/D	S/U/A
AMTD	UE		D	S
		_	_	+
		_	_	+
			+	

Select/Omit criteria :

AND/OR	SELECT/	OMIT	Field	Operation(s)
SEL	ECT	CREE	DIT	_COMP(GT 100000)
AND		AM	TDUE_	COMP(GE 20000)

Note that since no entry was made in the SELECT/OMIT column for the second select/omit statement the 2 statements are ANDed together. In this case only customers with CREDIT greater than 100000 AND AMTDUE greater than or equal to 20000 are selected.

#### Example 5

Create a logical view/file called CUSMSTS3 that will order CUSMST by CREDIT. Only records with CREDIT greater than 100000 or AMTDUE greater than or equal to 20000 are to be visible via this view.

Logical view name : CUSMSTS3

Desc of logical view : High credit or large debt customers

Access path maint opt : IMMED Unique? NO Dynamic select? NO Key field details:

Field	Description	A/D	S/U/A
CREDIT	Г		_
			+
			+
		+	

Select/Omit criteria:

AND/OR SELECT/OMIT Field Operation(s) SELECT\_\_\_\_CREDIT\_\_\_\_COMP(GT 100000)\_\_\_\_\_ OR SELECT\_\_\_\_ AMTDUE\_\_\_ COMP(GE 20000)\_\_\_

Note that since an entry was made in the SELECT/OMIT column for the second select/omit statement the 2 statements are read together. In this case only customers with CREDIT greater than 100000 OR . AMTDUE greater than or equal to 20000 are selected.

#### Example 6

Create a logical view/file called CUSMSTS4 that will order CUSMST by STATE and CUSTNO. Only records with a STATE of NSW, VIC or QLD are to be visible via this view.

Logical view name : CUSMSTS4 Desc of logical view : East coast customers by state, cust Access path maint opt : IMMED Unique? NO Dynamic select? NO Key field details Field Description A/D S/U/A

STATE	
CUSTNO	+
	+

Select/Omit criteria:

AND	/OR	SELECT/	OMIT	Field	Operation(s)
	SELI	ECT	STAT	ГЕ	COMP(EQ 'NSW')
OR	SE	LECT	ST	ATE	COMP(EQ 'VIC')
OR	SE	LECT	ST	ATE	COMP(EQ 'QLD')

An identical condition can be made using the VALUES keyword: Select/Omit criteria

AND/OR SELECT/OMIT Field Operation(s) SELECT\_\_\_\_\_ STATE\_ VALUES('NSW' 'VIC' 'QLD')

#### Example 7

Create a logical view/file called CUSMSTS5 that will order CUSMST by CUSTNO. Only records with an ACCTYP value of 2, 3 or 4 are to be visible via this view.

Logical view name : CUSMSTS5 Desc of logical view : Customers with type 2, 3 or 4 accounts Access path maint opt : IMMED Unique? NO Dynamic select? NO Key field details:

Field Description A/D S/U/A CUSTNO				
+				
+				
Select/Omit criteria:				
AND/OR SELECT/OMIT Field Operation(s)				
SELECT ACCTYP COMP(EQ 2)				
OR SELECT ACCTYP COMP(EQ 3)				
OR SELECT ACCTYP COMP(EQ 4)				
The select /omit condition could also be expressed as:				
SELECT ACCTYP COMP(GE 2)				
AND ACCTYP COMP(LE 4)				
or:				

SELECT\_\_\_\_ ACCTYP\_\_\_ VALUES(2 3 4)\_\_\_\_

or:

SELECT\_\_\_\_ ACCTYP\_\_\_ RANGE(2 4)\_\_\_\_

↑ 4.3 Logical View Development

## 4.3.5 Select/Omit Concepts

For RDMLX Files, the use of Select/Omit Criteria is not recommended. Instead, use the WHERE parameter to limit the data retrieved.

Select/Omit Criteria can be entered to specify details of any records that are to be either selected or omitted from a logical view. If a record is omitted, it means that although the record is present in the physical file, it is seemingly invisible in the logical view. A record will be omitted if it fails the selection test or passes the omission test. The criteria are entered as a series of statements.

A record can be omitted for 2 reasons:

- It fails to match the selection criteria.
- It matches the omission criteria.

Select/omit criteria is a powerful facility but should be used with some caution because the effect of making some records in the physical file "invisible", when accessed via certain logical views, can cause confusion amongst some users of the system.

Select/omit criteria are entered as a series of select/omit statements. When both SELECT and OMIT are used the order of the statements is important. The statements are processed in the order specified. If a record matches the criteria of a statement it is selected or omitted as specified and following statements are not tested.

Note that the type of record select/omit logic used by LANSA is identical and totally compatible with that used in IBM i DDS (data description specifications). In fact the select/omit criteria specified under LANSA are directly translated into DDS select/omit statements.

The only difference is that, under LANSA, the full words SELECT or OMIT are used. In DDS, they are indicated by entering an "S" or "O" into column 17 of the input form. Other than this difference, they are absolutely identical in specification logic.

The type of checking done by LANSA to the select/omit statements entered is relatively unsophisticated. As such the possibility arises of invalid select/omit statements being accepted into the system.

If this happens the problem will manifest itself the first time that an attempt is made to create the logical file. The LANSA job will fail and messages on the job log will indicate that the logical file failed to create successfully. If this happens, examine all source listings named QDDSSRC produced by the job.

Locate those which contain errors. If the errors indicate problems with the DDS select/omit statements generated by LANSA then delete and redefine the associated logical file so that the LANSA select/omit statements are acceptable to the DDS processor.

Since the LANSA select/omit statements are totally compatible with those used in DDS it would be beneficial to read the sections in the IBM supplied manual "Data Description Specifications" that relate to select/omit processing and the select/omit keywords CMP, COMP, RANGE, VALUES and ALL.

#### Also See

Select Omit Criterion in the Technical Reference Guide

1 4.3 Logical View Development

# 4.4 Virtual Field Development

A virtual field is a special feature of the LANSA Repository that allows a field value to be dynamically derived based on other fields in the file. To understand how to use virtual fields, review the following topics:

- 4.4.1 Virtual Field Concepts
- 4.4.2 Distinction between real and virtual fields
- 4.4.3 Types of Virtual Fields
- 4.4.4 Examples of Virtual Field Applications
- 4.4.5 Input or Output Virtual Fields
- 4.4.6 Steps to Create Virtual Fields
- 4.4.7 Virtual Field Considerations

## Also See

Virtual Fields in File in the Technical Reference Guide

Field in File tab in the User Guide

1 4. Creating Application Databases

# 4.4.1 Virtual Field Concepts

A virtual field is a special feature provided as part of the LANSA Repository. Used properly, virtual fields can be very helpful in simplifying the processing of logic within applications and in improving the user interface of your applications.

A virtual field is a field which appears as part of a file but does not actually exist in the physical database file. Its value is dynamically derived based on the fields in the file. (Virtual fields can also be derived from other virtual fields.) Virtual fields provide flexibility, since you can combine or extract data from fields in a file and extend the information already contained in the file.

You may use virtual fields for implementing "conceptual structures" within the database. For example, you can take the fields CENTURY, YEAR, MONTH, DAY to create a single field called date. You could also store a date in YYMMDD format but retrieve and display it as MMDDYY or DDMMYY.

To a program, the field appears to exist as part of the file; however, the value is not actually stored but derived by the I/O Module when the record is input or output from the file. If you were to display the contents of the file, you would not see the virtual fields. Hence, virtual fields cannot be used as keys in a file. Some examples of using virtual fields:

- Display an Order Date in MMDDYY format but store the data as YYMMDD.
- Display the Discount Amount which is calculated as 75% of the Original Amount.
- Display the Telephone Number which is the Area Code and Local Phone Number.
- Display the First Name which is the first 10 characters of the Customer Name.

You can use virtual fields to simplify, standardize and centralize field related string handling, conversions and calculations.

A virtual field must be defined in the repository and may have field and file level validations associated with it. Also, validation rules for the fields used in defining the virtual field are still enforced. A virtual field can be used like any other field in the repository.

Virtual fields are not part of the real file and cannot be directly used outside of LANSA. Virtual fields are made possible using LANSA's object access

module. Client applications using LANSA Open and any applications which use the LANSA Object Access Module can access the LANSA virtual fields.

Virtual fields are derived and are not real fields in the file. Virtual fields cannot be used as a key field for a physical file or a logical view. If you require fields for these purposes, you will need to make them into real fields as opposed to virtual fields.

LANSA provides several methods of defining virtual fields:

- LANSA extended definitions such as Date Conversion, Substring, Concatenation, Mathematical Calculation, Code Fragment (refer to Virtual Fields in File in the *Technical Reference Guide*)
- RPG or C program coded definitions (refer to Virtual Field Derivation in the *Technical Reference Guide*)
- database trigger functions can also be used to create virtual fields (refer to 4.5.6 Trigger Concepts).

RPG or C coded definitions may be necessary for complex actions such as type and length conversions, date to age conversions, or dynamic data conversion.

Also, review the 4.7.1 Predetermined Join Field Concepts for details about another form of virtual field that may use field values from other files.

## Also See

4.4.2 Distinction between real and virtual fields

4.4.4 Examples of Virtual Field Applications

1 4.4 Virtual Field Development

# 4.4.2 Distinction between real and virtual fields

The distinction between the real and virtual fields can be best illustrated by example.

Consider an accounting file called ACCMST that contains an 8 digit field called CHTACC (chart of accounts) that looks like this:

This example is for chart of accounts number 08634312. In actual fact the chart of accounts has 3 components. These are:

- 1. A 2 digit company number (08),
- 2. A 2 digit division number (63), and
- 3. A 4 digit cost center (4312).

Since the company number, division number and cost center can be "derived" directly from the chart of accounts number it would be possible to define in file ACCMST three "virtual" fields called COMPNO, DIVNUM and COSTCT. If this was done the definition of file ACCMST would contain:

Field	Alphanumeric Length
A "real" field called CHTACC	8
A "virtual" field called COMPNO	2
A "virtual" field called DIVNUM	2
A "virtual" field called COSTCT	4

Of course there is a missing link. That is the definition of how virtual fields COMPNO, DIVNUM and COSTCT are "derived" from the real field CHTACC. There are two methods available that allow you to specify how a virtual field is derived:

• The first method is the Extended definition of virtual fields function, which allows you to perform the most commonly used actions to derive virtual fields from real fields and vice versa.

• The second method is achieved by entering program code. This method is by far the most flexible method of specification for more complex actions but has platform implications for code.

In this example, the virtual fields COMPNO, DIVNUM and COSTCT could have been derived by using the "Extended definition of a virtual field" function by Substringing the real field into CHTACC OR by entering a few lines of RPG/400 code that mapped field CHTACC into fields COMPNO, DIVNUM and COSTCT.

Once this has been done COMPNO, DIVNUM and COSTCT would appear to be in file ACCMST. However, if the file was examined it would be found that the fields do not actually exist. Hence the name "virtual" fields.

Note that fields COMPNO, DIVNUM and COSTCT are only used when reading (i.e. inputting) from the ACCMST file. When writing to or updating the ACCMST file they have no particular meaning.

This need not be so, in fact virtual fields COMPNO, DIVNUM and COSTCT could be used to "re-assemble" field CHTACC when outputting to the file. This feature of virtual fields is extremely useful in some situations.

## Also See

- 4.4.4 Examples of Virtual Field Applications
- 4.4.1 Virtual Field Concepts
- 1 4.4 Virtual Field Development

# 4.4.3 Types of Virtual Fields

LANSA provides two methods of defining virtual fields:

- LANSA extended definitions
- RPG or C program coded definitions.

LANSA's extended definitions allow you to perform:

- Substringing. For details, refer to Substring in the *Technical Reference Guide*.
- Concatenation. For details, refer to Concatenation in the *Technical Reference Guide*.
- Mathematical calculations. For details, refer to Mathematical Calculations in the *Technical Reference Guide*.
- Date conversion. For details, refer to Date Conversion in the *Technical Reference Guide*.
- Code Fragment. For details, refer to Code Fragment in the *Technical Reference Guide*.

RPG or C coded definitions (refer to Virtual Field Derivation in the *Technical Reference Guide*) may be necessary for complex actions such as:

- type and length conversions
- date to age conversions
- dynamic data conversion.

LANSA trigger functions may also be used to create virtual fields. Trigger functions use RDML and provide an alternate option to using RPG or C for coding the virtual field logic.

A virtual field must be defined in the repository and may have field and file level validations associated with it. Also, validation rules for the fields used in defining the virtual field are still enforced. A virtual field can be used like any other field in the repository.

Virtual fields are not part of the real file and cannot be directly used outside of LANSA. Virtual fields are made possible using LANSA's object access module. Client applications using LANSA Open and any applications which use the LANSA Object Access Module can access the LANSA virtual fields.

## Also See

4.4.2 Distinction between real and virtual fields

4.4.1 Virtual Field Concepts↑ 4.4 Virtual Field Development

# 4.4.4 Examples of Virtual Field Applications

The 4.4.2 Distinction between real and virtual fields section described a very simple example of three "virtual" fields that were directly derived from a "real" field by simple mapping.

These examples demonstrate some of the capabilities of the virtual field facility and common examples of its usage. Hopefully it can also be seen that the capabilities are only really limited by the imagination of the user:

## Substring and concatenation

The example in the preceding section that involved breaking a field down into three virtual fields is an example of substring. Virtual fields can be used in all forms of substring. This includes substringing one real field into multiple virtual fields and substringing multiple real fields into one virtual field (also called concatenation).

## Access to files with no external description

Many older S/38 applications, and S/36 migrated IBM i applications, use files that have no "external" field descriptions. These are also referred to as "internally described" files. These files appear as if they only contain one very large field which is in fact the entire file record, rather than a series of fields.

In such cases, all the fields in the file can be defined as virtual fields and then mapped from the record. This is in fact another example of substringing one real field into multiple virtual fields.

This case would also involve the mapping of virtual fields to the real field before output to the file. For more details, refer to examples 1 and 2 in Examples of Virtual Fields & Derivation Code in the LANSA for i User Guide.

## Simplification and standardization of common data manipulations

In some applications the manipulation of data from a file in a particular way is very common and is repeated many times in differing applications.

For instance, consider an inventory file. The rule to determine whether a product requires re-ordering may be something like ("quantity on hand" + "quantity awaiting return" - "quantity on order") < "re-order level". This rule may be used in many different applications, particularly in the inquiry and reporting areas.

The logic in this rule can be simplified and standardized by using a virtual field. If a virtual field called REORDER was set up in the file definition, then

derivation code could be written to evaluate the rule and set REORDER to "YES" or "NO".

This approach has several advantages:

- Simplification. To users of the file there appears to be a field called REORDER which can be very simply tested as "YES" or "NO".
- Standardization. The rule to determine the re-order flag is standardized. There is no chance of slightly different rules being used in different applications.
- Centralization. The rule is held in one place only, not repeated in many different applications. If the rule changes it need only be changed in one place.

## Type and length conversions

When the type (i.e. alpha, packed or signed) or length of a field is inappropriate or inconvenient for common usage requirements a virtual field can be established.

For instance a 15 digit numeric field that never contains more than 999 can annoy users in reporting environments because it wastes 12 spaces on the report. A virtual field containing only 3 digits can be set up for use on reports.

Likewise an alphanumeric date may annoy users because it cannot be printed with an edit code/word. A numeric virtual field could be set up to solve this problem.

## Aggregation and accumulation

Many database files contain "arrays". This most commonly takes the form of a series of fields like SAL01, SAL02, SAL03 ...... SAL12 representing company sales for each of the months of the year.

Virtual fields can be used to aid users when working with these type of structures, particularly in the reporting and inquiry environments.

For instance virtual field SALYR could be defined to contain the total of fields SAL01 -> SAL12.

Similarly virtual fields SALQ1 (containing total of SAL01 -> SAL03), SALQ2 (containing total of SAL04 -> SAL06), etc, could be defined to contain quarterly sales totals.

#### **Date conversion**

When a date is held in a real field in format YYMMDD it can be easily mapped into a virtual field in format DDMMYY. The real field is the most commonly used format for ordering the file, but the virtual field format is the most commonly used for printing.

#### Date to age conversion

A date (possibly of date of birth) field in a file can be converted into two virtual fields containing "age in years" and "age in months". Note that derivation logic here would actually involve the date field in the file and the current date (which is why the "age" cannot be stored in the database file - it will be wrong tomorrow).

Note that the logic involved here is more complex. In fact many sites would have a subroutine to do the job. Since derivation logic is specified to LANSA as RPG code it is easy to call existing subroutines.

#### Dynamic data conversion

Just about any form of dynamic data conversion can be achieved by using virtual fields.

To take an example to the extreme, imagine that a new company requirement for printing customer names is that:

- They are always in uppercase
- All quotes, commas, full stops and hyphens are removed
- The name must be reversed (i.e. printed backwards)

If the customer name field was called CUSNAM then it would be a simple task to define a virtual field in the file called PRTNAM that matched these requirements.

## Also See

4.4.2 Distinction between real and virtual fields

- 4.4.1 Virtual Field Concepts
- 1 4.4 Virtual Field Development

# 4.4.5 Input or Output Virtual Fields

When using virtual fields in LANSA it is very important to understand and properly define when the virtual field is derived. For example, you can use virtual fields as output only to display information, such as a date in a different format. Or, you can use virtual fields as input fields which will determine what value is written to the real field in the file. Understanding when a virtual field is derived becomes even more important once predetermined join fields and triggers are introduced. A field can be specified as both on input and before output.

Derive value when record is read = DISPLAY ONLY

The "derive value when record is read" option means that the virtual field is created when you read the information from the file. The information will now be derived and available for display. For example, a date is stored as YYYY/MM/DD so the file can be properly sorted by date, but the information is displayed to the user as DD/MM/YY. This virtual date field would be derived on input, in other words, when you read from the file.

Populate real field when writing to file = UPDATE FILE

The "populate real field when writing to file" option means that information entered in a virtual field will be written out to the file field(s) used to define the real fields. The virtual fields are the user input fields and will write information to a real field in the file. For example, the user enters a date in DD/MM/YY and it is then converted and stored in the file as YY/MM/DD.

To summarize:

- If a virtual field is derived when read, it will be used for display purposes.
- If a virtual field is derived when writing to file, it will be used to update the file.
- If a virtual field is derived for both, it will be used for both display and update purposes.

## Also See

Derive value when record is read in the *Technical Reference Guide* Populate real field when writing to file in the *Technical Reference Guide* 1 4.4 Virtual Field Development

# 4.4.6 Steps to Create Virtual Fields

Virtual fields are part of a file definition in the repository and are accessed while working with a specific file. Remember that the virtual field, like any field in a file, must be defined to the repository before you add it to a file. If you follow the recommended naming standards, virtual fields are easily identified as they are nine characters long (xxxxxVIR). (Refer to Field Naming Standards.)

If the virtual field is derived using code, you will need to indicate this using the appropriate input field/option. For the IBM i, virtual code is written in RPG while C is used for other platforms such as Windows.

It is recommended that you review the 4.4.7 Virtual Field Considerations before using virtual fields.

The procedure for creating a virtual field using the extended definition is as follows:

- 1. Check that the field is defined in the repository.
- 2. Add the field to the file as a virtual field.
- 3. Select the type of virtual field.
- 4. Add the details of the virtual field.

Once defined to the file, you may add validation rules for the virtual fields.

## Also See

Virtual Field Definition in the *Technical Reference Guide* Field in File tab in the *User Guide* 

1 4.4 Virtual Field Development

# 4.4.7 Virtual Field Considerations

Following are some important considerations when using virtual fields:

- More complex virtual fields can also be defined using 3GL code. The 3GL code will be written in RPG for the IBM i and in C for Windows (other platforms). If you are building applications which will execute on more than one platform, you must be very careful with 3GL code. For more information about creating virtual fields using 3GL code, please refer to Virtual Fields in the *LANSA for i User Guide*.
- Remember to follow the field naming standards so that virtual fields will end in a 3 character suffix of "VIR". This will help developers identify a field as a virtual field. (Refer to Field Naming Standards.)
- Virtual fields are derived and are not real fields in the file. Virtual fields cannot be used as a key field for a physical file or a logical view. If you require fields for these purposes, you will need to make them into real fields as opposed to virtual fields.
- For the most part, LANSA treats the virtual fields like any other field in the file. You can add field and file level validation rules to a virtual field. If the file is being used for output, the validation rules should be specified on the virtual field and not the real field. You may encounter difficulties if you have validation rules on the real field but use virtual fields as before output to write to the file.
- A predetermined join field is a special type of virtual field. (Refer to 4.7 Predetermined Join Field Development.)
- You can also use trigger functions to create virtual fields. (Refer to 4.5 File Rules & Triggers Development.)

## Also See

Virtual Field Definition in the *Technical Reference Guide* Field in File tab in the *User Guide* 

1 4.4 Virtual Field Development

# 4.5 File Rules & Triggers Development

LANSA recommends that rules and triggers be defined at the file level in the repository. As an introduction to rules and triggers, review the following topics:

- 4.5.1 Rule Concepts
- 4.5.2 Rule Types
- 4.5.3 Rule Hierarchy
- 4.5.4 Understanding Rule Sequence
- 4.5.5 What is Database Referential Integrity?
- 4.5.6 Trigger Concepts
- 4.5.7 Trigger Points When
- 4.5.8 Trigger Conditions If
- 4.5.9 Trigger Functions
- 4.5.10 Trigger Design Considerations
- 4.5.11 LANSA Database Triggers

You may also wish to review Field Rules and Triggers Development.

## Also See

Rules and Triggers Definitions in *Technical Reference Guide* File Rules and Triggers Tab in *User Guide* 

1 4. Creating Application Databases

# 4.5.1 Rule Concepts

Within LANSA, there are three levels at which business rules can be defined:

- Field Level as part of the repository field definition
- File Level as part of the repository file definition
- Program Level as defined in the code of a Function or Component.

LANSA supports repository-based rules in order to centralize the business rules of your application. Instead of repeatedly coding the same business rules into each program, LANSA allows a rule to be defined once for a field or file, and then it will be applied during database operations from any LANSA application. To simplify your application development, LANSA supports a variety of different 4.5.2 Rule Types.

The order in which validation rules are performed is Function, then Field and lastly File. Your functions may perform coded checks before they perform database operations. The field and file validation rules are not called until the function performs a database operation using the Object Access Module (OAM).

The order of importance of the validation hierarchy is defined as Field, then File and lastly Function. For a field to be used in a file, it must be defined to the repository first. If the same field is used in three different files, all three files will all refer to the same repository field. Consequently, each file will use the same field level validation rules for that specific field. (This logic is also true of functions which use one file.) Hence, field rules are included in files which are then used by functions. Even though a function level check may be performed before the field and file checks, the field and file checks are considered higher in the hierarchy because they provide ultimate control of your validations.

## Also See

Field Rules and Triggers Development
Rule and Trigger Definitions in the *Technical Reference Guide*4.5.2 Rule Types
4.5.3 Rule Hierarchy
4.5.4 Understanding Rule Sequence
1 4.5 File Rules & Triggers Development

# 4.5.2 Rule Types

The following rule types are supported at the field and file level:

- Range of values rule: Allows a field to be checked against various ranges of values. For instance, an Australian postcode should be in one of the ranges 2000 2999, 3000 3999, etc. Refer to Range of Values Checks in the *Technical Reference Guide*.
- List of values rule: Allows a field to be checked against a list of values. For instance, an Australian state should be in the list QLD, NSW, VIC, etc. Refer to List of Values Checks in the *Technical Reference Guide*.
- Code/table file lookup rule: Allows a field to be "looked up" in a code file or table. For instance, a product number may be looked up in the product master file to check that it is a valid number. Refer to Code File/Table Lookup Checks in the *Technical Reference Guide*.
- Simple logic rule: Allows simple conditions to be evaluated to check a field. For instance, "item weight must be less than (item volume \* 10.643)" may be a check used in an order entry system. Refer to Simple Logic Check in the *Technical Reference Guide*.
- Complex logic rule: Allows complex validation checking to be performed by your own LANSA functions or 3GL application programs. For instance, the validation of a "due date" may be done via a function or program that can account for public holidays, weekends, etc. Refer to Complex Logic Check in the *Technical Reference Guide*.
- Date format/range rule: Allows a field to be validated as a date in a certain format (DDMMYY, MMDDYY, etc.) and in a certain range. For instance, a "date order due" may have to be in format DDMMYY and no more than 90 days into the future. Refer to Date Format/Range Check in the *Technical Reference Guide*.

In addition to these rule types, you may use Triggers, in the *Technical Reference Guide*, to add validation rules to your applications.

#### Also See

Rule Definitions in the Technical Reference Guide

↑ 4.5 File Rules & Triggers Development

## 4.5.3 Rule Hierarchy

Rules and triggers exist in a 3 level hierarchy:

# FIELD LEVEL | | FILE LEVEL | | PROGRAM LEVEL

- **Field level rules/triggers** always apply to a field under all circumstances. For instance, a field level rule that says a post code must be in the range
- 2000 to 2999 will ensure that no matter when or how the post code field is used it will ALWAYS have to be in the range 2000 to 2999. If post code is used in FILE1, FILE2 and FILE3, all files will have the same rule applied.
- **File level rules/triggers** apply only to the field within the file. For instance, FILE1 may have a rule that says field DATDUE must be in the format DDMMYY, however FILE2 may have a check that says field DATDUE must be in the format YYMMDD. This may be perfectly valid.
- **Program level rules** apply only to the field as used within a program (Function or Component). For instance, a Function named ORDERS may add the rule that field DATDUE (used in previous example) must be within the next 30 days. This check will be done in addition to the file level checks associated with file FILE1.

Almost all validations will be specified at file level. Field level validations apply everywhere a field is used so you must be careful that the rule is truly a "global" rule to an application. (Refer to Field versus File Level Rules.)

## Also See

- 4.5.1 Rule Concepts
- 4.5.4 Understanding Rule Sequence

## 4.5.4 Understanding Rule Sequence

Rules are performed and displayed based on their Rule Sequence number (within the validation level). Sequence numbers must be unique within the validation level.

For instance these entries would be performed in the sequence RULE04, RULE01, RULE02, RULE05, RULE03:

Rule	Level	Sequence
RULE01	Field	2
RULE02	Field	3
RULE03	File	2
RULE04	Field	1
RULE05	File	1

#### Also See

4.5.1 Rule Concepts
4.5.3 Rule Hierarchy
Rule Definitions in the *Technical Reference Guide*↑ 4.5 File Rules & Triggers Development

## 4.5.5 What is Database Referential Integrity?

Referential Integrity is an important feature within an application database design. Referential Integrity refers to the ability of the database to keep related data from being corrupted. The data references between files must remain intact. For example in the Personnel System demonstration application database, all of the records in the Section file require a Department Code. These Department Codes must exist in the Department file. If this reference is lost, the database is considered to be corrupted and referential integrity is broken.

Referential Integrity means that two specific database rules must exist between files:

- 1. Do not delete data required by another file. You must check that there are no records in related files that require the record which you are deleting.
- 2. Do not add incomplete data to a file. You must check that the related information exists in another file before adding information to a file.

These two rules describe data dependencies from the perspective of each file involved in the relationship. Referential integrity works in two directions and is sometimes described as "forward" and "backward" referential integrity. This concept is similar to access routes. File1 has a relationship to File2, and File2 has a relationship to File1. In order to build referential integrity into your database, you must have a good understanding of the application business rules.

LANSA implements database referential integrity using file rules in the repository so that your application is independent of the database used. Independence is a major advantage when deploying your application.

## Also See

Personnel Demonstration ApplicationReferential Integrity Example↑ 4.5 File Rules & Triggers Development

# **Referential Integrity Example**

In the Personnel System demonstration application database, there is an example of the Department, Section and Employee files. For these files, the rules required are as follows:

- 1. A Department Code cannot be deleted from the Department file if there are any records in the Section file which require that Department Code. For example, you should not be able to delete department code ADM unless there are no records in the Section file with code ADM, otherwise all the Sections for ADM have no parent record.
- 2. Department Code cannot be used in the Sections file unless the Department Code exists in the Department file. For example, you cannot create a new section in a department XYZ unless XYZ exists in the Department file.
- 3. A Section Code cannot be deleted from the Section file if there are any records in the Employee file which require that Section Code. For example, you should not be able to delete section 01 unless there are no records in the employee file with section code 01, (otherwise the employee would not be able to refer to this record).
- 4. A Department and Section Code cannot be used in the Employee file unless the codes already exists in the Section file. For example, you cannot add an employee to department XYZ section 01 unless it already exists in the Section file.

To define referential integrity rules, you must understand the business application and the data model. For example, no referential integrity rules are needed between Department and Employee because Employee refers to the Section not the Department. Employee data is based strictly on the Section file which, in turn, is based on the Department file. For example, if you create a department ABC, an employee cannot work for that department ABC until a Section is created.

#### Also See

Personnel Demonstration Application

↑ 4.5.5 What is Database Referential Integrity?

# 4.5.6 Trigger Concepts

A database trigger is a condition that is defined at the field or file level in the repository. The trigger definition allows a function to be linked to a specific database operation and condition. The trigger function will be invoked **automatically** when a specific type of I/O operation occurs to a file and when a specific set of conditions are met.

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

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

Activity A Flag Order Historical Details

Activity B Print Outstanding Credit Invoices

Activity C Send a Message to the Sales Department

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

Similar to validation checks, LANSA supports repository-based triggers in order to centralize the business logic of your application. Instead of repeatedly coding the same business procedures into each program, LANSA allows a trigger to be defined once for a field or file, and then it will be applied during database operations from any LANSA application.

Sensible use of triggers may transform the way that an application is designed. The user interface can be fully designed, and then the complexities and rules can be introduced later by using file validation rules and database trigger functions. The resulting design is much more in the "object oriented" style. Triggers separate "business function" from "user interface" in a much clearer and easier way.

## Also See

Trigger Definition in the *Technical Reference Guide* Trigger Condition in the *Technical Reference Guide*  Trigger Functions in the LANSA for i User Guide↑ 4.5 File Rules & Triggers Development

## 4.5.7 Trigger Points - When

The trigger point defines when the trigger should be called. A trigger may be called at many different points.

#### Before and After Open/Close file:

Before and after open/close triggers are invoked immediately before or after an attempt is made to open or close a (logical) file.

## Before and After Read (Input):

Before read triggers are invoked immediately before an attempt is made to read a record from a file, and therefore have no access to information from that record. After read triggers are invoked just before the details are passed back to the invoking function

#### **Before Insert, Update or Delete:**

These before action triggers are invoked immediately before an attempt is made to insert a new record into a file, update an existing record in a file, or delete an existing record from a file.

#### After Insert, Update or Delete:

These after action triggers are invoked immediately after data manipulation attempts are made (as described above). The after update trigger has access to both the old and new contents of the record.

The trigger point is a very important consideration. When before action triggers are invoked, it is not guaranteed that the action will be successful. You must always consider possible failure.

#### Also See

Trigger Points in the *Technical Reference Guide* ↑ 4.5 File Rules & Triggers Development

# 4.5.8 Trigger Conditions - If

The use of a trigger condition is optional in LANSA. Trigger conditions provide fine grain event control so that a trigger is not always being called only to do nothing. For example, you may specify that a trigger is called only if an Employee's salary is greater than 100,000.

If you do not specify a condition, the trigger will be invoked unconditionally, that is every time the trigger points are reached. Using trigger conditions will greatly improve the performance of the application as the trigger is only called when it is needed.

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

Possible operations are GT, GE, LT, LE, EQ, NE and REF.

**Note:** The operation codes may have a suffix 'P', which indicates that the Previous value of the nominated field should be used. The value of a previous field can only be used in 'Update' operations. REF (Refers to) is only allowed on 'Read', 'Insert' and 'Update' operations.

There are no conditions allowed for Open and Close triggers.

## Also See

Trigger Condition in the Technical Reference Guide

1 4.5 File Rules & Triggers Development

## **4.5.9 Trigger Functions**

A LANSA trigger function is a special type of LANSA function, which is invoked automatically when a specific type of I/O operation occurs on a file and when a specific set of conditions are met.

There are 2 building block application templates available, designed to form the base of a trigger function: one for field-based triggers, and one for file-based triggers.

A trigger should encapsulate all the trigger rules for a particular field or file, (depending upon whether the trigger is linked at field level or file level).

For details about trigger functions, refer to Trigger Functions in the *Technical Reference Guide*.

1 4.5 File Rules & Triggers Development

## 4.5.10 Trigger Design Considerations

There are a number of design principles for trigger functions, as well as some rules for LANSA triggers. Following are some important design considerations for triggers:

## **Encapsulation:**

The trigger function should be designed to encapsulate all the rules into one, and only one, place. For example, if you design a trigger function to calculate payroll tax, then you should have all the calculations logic in one place only. This means that you only need to update one function when the "method" of calculation changes.

## **Deferment:**

The details of a "method" do not have to be defined, or even known about, during initial system design. This also means that a "method" can be introduced into an application design at any time. For instance, the payroll tax calculation method does not have to be defined first, before the applications that create or update employees are defined and tested. When the payroll tax trigger is created and defined, it will immediately begin to affect the processing of all existing applications.

## **Re-usability:**

The trigger must be designed bearing in mind that it is, (or might be in the future) reused by any application. In the payroll tax example the trigger could be activated from a normal IBM i NPT device via an "Employee Maintenance" function, or from a PC application via the LANSA Open facility.

## **Transparency:**

The fact that the calculate payroll tax logic is present and being used must be invisible and immaterial to an RDML builder creating an "Employee Maintenance" function.

## Separation of the "method" from the "event":

The trigger function must define what to do when an "event" happens (i.e: the "method"), but it does not have to detect the occurrence of the event. The occurrence of the event must be detected by the trigger points and conditions in the data dictionary.

## ↑ 4.5 File Rules & Triggers Development

# 4.5.11 LANSA Database Triggers

# What is a Database Trigger?

A trigger is a set of actions that are run automatically when a specified change or read operation is performed on a specified physical database file. The change operation can be an insert, update, or delete high level language statement in an application program. The read operation can be a fetch, get, or read high level language statement in an application program. As well as from high level languages, these operations could also originate with queries, ODBC, JDBC, certain CL commands for example RCVF, interactive SQL, embedded SQL or Call Level Interface SQL.

This is in contrast with LANSA triggers which are special LANSA functions that run automatically when a specified database operation occurs via LANSA. The implementation of database triggers with LANSA is subsequently referred to as "LANSA DB Triggers" or "Database Triggers".

Database Triggers are only implemented for database access occurring on IBM i in an RDMLX partition.

Before enabling LANSA DB Triggers for a file ensure that you thoroughly understand the potential implications for your existing applications.

Please refer to the following topics in this guide. You may also find the Database Trigger Architecture useful. It is also recommended that the section on "Triggering automatic events in your database" in the DB2 for IBM i Database Programming manual or in the IBM Infocenter, and the section on "Triggers" in the Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for IBM i red book be read and understood.

Enable Database Triggers for a file

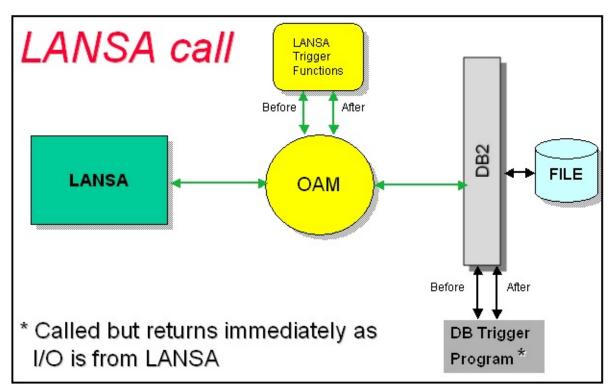
What happens when a Database Trigger is executed?

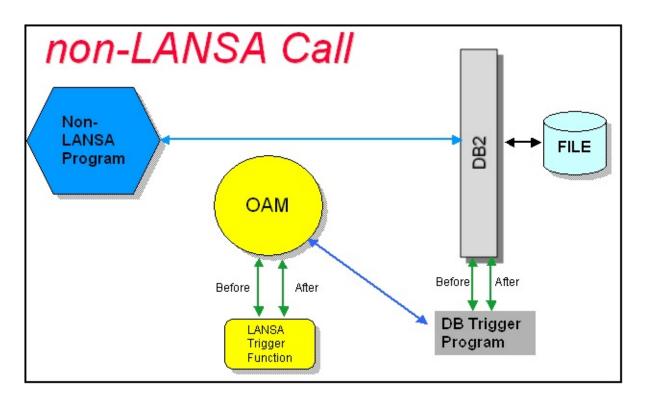
What is the database trigger environment user exit program?

What are the limitations with LANSA DB Triggers?

Other observations regarding LANSA Database Triggers

**Database Trigger Architecture** 





# **Enable Database Triggers for a file**

This makes the LANSA repository rules available to all I/O done against a file, whether made via LANSA or via non-LANSA means. For example, when a file is updated with an RPG or COBOL application, the validation rules within the repository will be enforced, and any LANSA Trigger functions will be executed. In addition, whenever a database trigger is triggered for a file, a trigger environment user exit program, DCTRGENVUE, must be in the job's library list so that the correct LANSA environment can be established. Database Triggers can only be enabled for a file in an RDMLX partition.

## What happens when I enable Database Triggers for a file?

When a file is enabled for Database Triggers, a Database Trigger program name is needed. This is initially defaulted to DB <filename>, (as described in Database File Trigger in the *Technical Reference Guide*) but can be overridden if this causes duplicate names with another OAM or Database Trigger program name. When the file is next made operational, a Database Trigger program is generated and compiled to run in a named activation group, any Database Trigger processing is added to the OAM, and any triggers added to the database, according to the validation rules and triggers defined for the file.

Insert	Before Insert	Before Insert
N/A	After Insert	After Insert
Update	Before Update	Before Update
N/A	After Update	After Update
Delete	Before Delete	Before Delete
N/A	After Delete	After Delete
N/A	Before Read	*
N/A	After Read	After Read
N/A	Before Open	*
N/A	After Open	*
N/A	Before Close	*

#### Validation rules and/or trigger functions result in DB Trigger

Note: \* means not implemented in the database

The Database Trigger program and database triggers are created in the I/O module library. The triggers are named:

\*

LANSA\_<program library>\_<partition>\_<library>\_<file>\_<trigger time>\_<trigger event>

where:

<program library>is the LANSA program library, for example DCXPGMLIB

<partition> is the LANSA partition, for example PRD

library> is the file library, for example DCXPRDLIB

<file> is the file name, for example PSLMST

<trigger time> is the trigger time, for example BEFORE

<trigger event> is the trigger event, for example INSERT

An example of an entire trigger name could be

LANSA\_DCXPGMLIB\_PRD\_DCXPRDLIB\_PSLMST\_BEFORE\_II

↑ 4.5.11 LANSA Database Triggers

## What happens when a Database Trigger is executed?

- 1. A database management program calls the DB Trigger program
- 2. If the I/O is from LANSA the DB Trigger program returns
- 3. If this is the first time for the program, the trigger environment user exit program (DCTRGENVUE) is called
- 4. The LANSA environment is established
- 5. The OAM is called to carry out any validation rules and / or call any LANSA trigger functions. In addition, for before insert and before update processing, the value of the special @@UPID field for LANSA defined files is enforced to be 1 for before insert and incremented by 1 for before updates. Also those two trigger time and events will cause stamp output attribute fields to be filled in.
- 6. If LANSA is configured to replace the library list on exit from LANSA (specified in the *Replace library list on exit from LANSA* option described in the Execution and Security Settings of the *LANSA for i User Guide*), the job library list is set back to what it was when the Database Trigger program was called. Otherwise the LANSA libraries are left in the job library list, and it is assumed that they will be there for subsequent calls of the Database Trigger program.
- 1 4.5.11 LANSA Database Triggers

#### What is the database trigger environment user exit program?

The database trigger environment user exit program is how the generated DB trigger program knows which LANSA system, partition, languages and date source the LANSA environment is to execute in. The database trigger environment user exit program is called DCTRGENVUE and must exist somewhere in the library list of any job that does I/O in any way to a file that has DB Triggers enabled.

The generated DB Trigger program will receive these parameters from the user exit program:

Description	Туре	Length	Usage
LANSA Program Library Name	*CHAR	10	Must be set
Partition	*CHAR	3	Must be set
Language	*CHAR	4	May be blank for a non-multilingual partition, otherwise must be set
Date format source	*CHAR	1	Must be set to S or J. Refer to the LANSA command parameters.

1 4.5.11 LANSA Database Triggers

## What are the limitations with LANSA DB Triggers?

 LANSA DB triggers only work correctly for files that are accessed via Native I/O.

This excludes:

- RDMLX Other Files that are only accessed via SQL, as well as
- LANSA files containing one or more LOB fields or fields that are promoted to being stored as LOBs in the database.
- The IBM i DB2 UDB for IBM i Database Programming manual warns "An SQL update operation involves a simultaneous read operation followed by a write operation. Read triggers will not be run for SQL update operations. An update trigger should be specified to cover this read followed by a write operation." The implementation of LANSA DB Triggers will simply cause the update processing to occur in this situation.
- No virtual field processing or validation will occur as part of LANSA DB Triggers. In particular, LANSA trigger functions will not have available to them the values of any virtual fields or PJF.
- Validation rules and LANSA trigger functions that are defined on a "where used" basis for real fields will always be executed when called via LANSA DB Triggers.
- ↑ 4.5.11 LANSA Database Triggers

## Other observations regarding LANSA Database Triggers

- Database Triggers are always called for example if a file that is enabled for Database Triggers has an after read trigger function defined for it, that after read trigger function is called even when using the IBM i command DSPPFM for that file. Another example is that CLRPFM is not allowed for files with before delete or after delete database triggers.
- Execution of LANSA database triggers is only available in RDMLX partitions. LANSA files with LANSA Database Triggers enabled cannot be imported into an RDML partition.
- Export of IBM i Other files (the physical and any logicals) as non-LANSA objects will require that the database triggers be removed manually before the export is run, and re-added after the export is completed.
- Care should be taken before enabling LANSA Database Triggers for a file that is used in existing 3GL applications, as the existing 3GL applications may be coded in such a way that if a Database Trigger failure is signaled back to them, they may simply abort and leave the database in a half updated condition.
- After read Database Triggers will be executed as a result of a keyed RPG SETLL, SETGT, C API \_Rlocate and the COBOL equivalent operations. This is a design decision in the DB2 for IBM i database.
- There are some restrictions to what can be done to a file when it has DB triggers attached and enabled, for example a file with before or after delete triggers cannot be the subject of CLRPFM, and a file with before or after insert triggers cannot be the subject of INZPFM. For more information on such restrictions, refer to the section on "Triggers and their relationship to other IBM i functions" in the DB2 for IBM i Database Programming manual or the equivalent section in the IBM InfoCenter.
- After turning on LANSA Database Triggers for a LANSA file, performance from LANSA will be affected. This is unavoidable as the database itself has no way of detecting that the Database Trigger should not be executed in this situation. The Database Trigger program will be executed, determine that it has been executed as a result of an I/O from LANSA, and return.
- Because LANSA Database Trigger programs execute in a named activation group, care must be taken if commitment control is used. It is strongly recommended that the section on Commitment Control in the IBM InfoCenter be read and understood before LANSA Database Triggers are

enabled. In particular, if the execution of the LANSA Database Trigger program or anything it calls results in database inserts, updates or deletes to files under commitment control, and activation group level commitment definitions are used then the effect on the existing application must be tested in a variety of situations to ensure that the application's integrity is preserved.

1 4.5.11 LANSA Database Triggers

## 4.6 Access Route Development

Changing a group of files into an application database requires the definition of file relationships or access routes. Before you begin to define your access routes, it is recommended that you have defined 4.3 Logical View Development for the files in your database.

To understand how to use LANSA's access routes, review the following topics:

- 4.6.1 Access Route Concepts
- 4.6.2 Access Route Examples
- 4.6.3 Access Routes in Personnel System Example
- 4.6.4 Access Route Considerations

## Also See

Access Route Definitions in the Technical Reference Guide Access Route Tab in the User Guide

1 4. Creating Application Databases

## 4.6.1 Access Route Concepts

LANSA uses access routes to describe relationships between files in a database. They provide information about the database map or schema. Access routes are simply text descriptions of file relationships in the database. They have no physical impact on the database.

An access route defines to LANSA a route from one file to another file. An access route answers the question: Given that I have a record from file A..... how do I access the associated record(s) in file B?

File A is always the file definition that is currently being worked upon. File B can be any other physical or logical file defined using the LANSA system.

File A can have many access routes. They could all be from file A to file B, or some could be to any other file that is related to file A. Access routes are defined in both directions, i.e. from file A to B and from file B to A.

One access route can only define the route from file A to file B. However, file A can have many access routes associated with it. Thus file A can have additional access routes to file C and to file D, etc., etc.

In addition, access routes can be followed in a chain. Thus if there is an access route from file A to file B, and an access route from file B to file C, then it is possible to start with a record from file A and locate the associated record(s) in file C.

Access routes are very important to the automatic process definition components of LANSA because:

- Users do not need to understand the data base structure. You only need to nominate a "base" or "starting" file and LANSA can then follow the access route chains to present the other accessible files as simple alternatives.
- The relationship between records in the 2 files in any access route is predefined as "1 : 1" or "1 : many". This enables the required screen formats to be automatically designed for the type of information expected.

Access routes are used in the following ways in LANSA:

- By the LANSA application templates to show related files
- By predetermined join fields to define cardinality and the links between files
- By LANSA Client to show linked or joined files.

### Warning

• Access routes only point to the active version of a file. This is the most

recently compiled version, and not necessarily the latest version. Similarly, a new file must be compiled at least once for an access route to point to it.

#### **Tips & Techniques**

• Do not confuse access routes with IBM i database access paths. Access routes have no relationship to access paths.

#### Also See

- 4.6.2 Access Route Examples
- 4.6.4 Access Route Considerations
- 1 4.6 Access Route Development

## 4.6.2 Access Route Examples

These examples apply to defining an access route within a file definition. Four related file definitions are used in these examples. These are defined as follows:

File:		ORDHDR	ORDLIN	CUSMST	PROMST
Desc:		Order header	Order lines	Customer master	Product master
Fields:		ORDNUM CUSTNO DATE	ORDNUM ORDLIN QUANTY PRODNO	CUSTNO NAME ADDR1 ADDR2 ADDR3	PRODNO DESC PRICE TAXRTE
<b>Primary</b>	1	ORDNUM	ORDNUM	CUSTNO	PRODNO
Key	2		ORDLIN		
Logical Views:		ORDHDRV1O	RDLINV1		
by key	1	CUSTNO	PRODNO		

To allow complete traversal of this database by users the following access routes would be defined:

Definition Name	Access Route Name	File to be Accessed via this Route	Max Rcds Expected	Action To Be Taken If No Records Found	Key Fields/Va Used For Access
ORDHDR	ORDHRT1	ORDLIN	99	IGNORE	ORDNUN
	ORDHRT2	CUSMST	1	ABORT	CUSTNO
ORDLIN	ORDLRT1	ORDHDR	1	ABORT	ORDNUN

	ORDLRT2	PROMST	1	ABORT	PRODNC
CUSMST	CUSMRT1	ORDHDRV1	9999	IGNORE	CUSTNO
PROMST	PROMRT1	ORDLINV1	9999	IGNORE	PRODNC

From this table it is easy to see how LANSA can follow the access route chains. For example, if the user nominated PROMST (product master) as a "starting" or "base" file, then the following would occur:

- Using access route PROMRT1 a route to ORDLIN (order lines) is available (via logical file ORDLINV1) which will access all invoice lines that use the product.
- From ORDLIN the associated order header (ORDHDR) details can be picked up using access route ORDLRT1. The access route ORDLRT2 back to PROMST would be ignored as it forms a "loop".
- From ORDHDR the associated customer details (CUSMST) can be found via access route ORDHRT2. Again ORDHRT1 would be ignored as it forms a "loop".

1 4.6 Access Route Development

## 4.6.3 Access Routes in Personnel System Example

Consider the Department, Section and Employee files which are part of the Personnel System Application database.

For the Department file, access routes to both the Employee and Section files are created. The access routes tell LANSA how the Department file will access the other files:

- Department will access the Employee file by using a logical file over PSLMST which is keyed by DEPTMENT. When Department accesses Employee, it will expect to find many records since a Department can have many Employees.
- Department will access the Section file by using the physical file with the DEPTMENT field as the key. When Department accesses Section, it will expect to find more than one record since a Department can have many Sections.

For the Section file, a relationship back to Department and a relationship to Employee are needed.

- Section will access the Department file by using the physical file with the DEPTMENT field as the key. When Section accesses Department, it will expect to find just one record since a Section belongs to just one Department
- Section will access the Employee file by using a logical file over PSLMST which is keyed by DEPTMENT and SECTION. When Section accesses Employee, it will expect to find many records since a Section can have many Employees.

For the Employee file, a relationship back to both the Department and Section files are needed

- Employee will access the Department file by using the physical file with DEPTMENT field as the key. When Employee accesses Department, it will expect to find just one record since a Employee works in just one Department at a time.
- Employee will access the Section file by using the physical file with SECTION field as the key. When Employee accesses Section, it will expect to find just one record since a Employee works in just one Section at a time.

#### Also See

Personnel Demonstration Application

1 4.6 Access Route Development

## 4.6.4 Access Route Considerations

Following are some important considerations when defining access routes:

- Access routes should always be defined in both directions, i.e. from file A to B and from file B to A.
- The Maximum Records parameter is important as it defines the cardinality of the file relationships. This parameter specifies the number of records that are expected to be found in the Accessed File that have a key matching the Key Fields/Values. This value is only significant in that it is one or more than one. If it is one, then a 1:1 relationship between the files is established, (for example, Employee works in a Department). If it refers to more than one, then a 1:many relationship is established. (for example, a Department has many Employees).
- Remember that access routes work both ways. If there is an access route from FILE1 to FILE2, then there should also be a corresponding access route from FILE2 to FILE1. For example, Department file has a 1:M relationship to the Employee file and the Employee file will have a 1:1 relationship to the Department file.
- When dealing with multiple parent-child relationships, it can be valuable to include access routes from the grandparent to the grandchild. These type of links are particularly useful to reporting tools like LANSA Client which can create a drill down report by following the access routes.
- Access routes are very useful for both developers and end-users, in the case of LANSA Client. It is important to take the time to add access routes to your application databases.
- Access routes can be used with Other files. They can be used to document the file relationships within your existing application databases.

1 4.6 Access Route Development

## 4.7 Predetermined Join Field Development

Predetermined Join Fields are a special type of virtual field in the LANSA Repository that uses fields from related files. To understand how to use Predetermined Join Fields, review the following topics:

- 4.7.1 Predetermined Join Field Concepts
- 4.7.2 PJF Examples
- 4.7.3 PJF Example using Personnel System
- 4.7.4 Steps to Create a PJF
- 4.7.5 Access Routes and PJFs
- 4.7.6 PJF Considerations

#### Also See

Predetermined Join Field Definitions in the Technical Reference Guide

Access Route Tab in the User Guide

1 4. Creating Application Databases

## 4.7.1 Predetermined Join Field Concepts

A Predetermined Join Field or PJF is a special feature of the LANSA Repository. A PJF is a special kind of virtual field. (Refer to 4.4.1 Virtual Field Concepts.) Unlike the other types of repository virtual files, a PJF is a virtual field whose value is determined by a value or values in another file. (Other virtual fields are always based on fields within its own file definition.) PJFs use access routes to determine the file relationships.

A Predetermined Join Field is used to hold the result of various operations performed on information retrieved from the file nominated by the access route. The field must be defined in the LANSA Repository.

The type of operations available to be performed depend on the relationship between the files as indicated by the number of records expected on the access route definition.

When the relationship is 1:many (refer to access route Maximum Records), the operations are TOTAL, MAXIMUM, MINIMUM, AVERAGE, and COUNT. (Also refer to PJF Type in the *Technical Reference Guide*.)

When the relationship is 1:1 using the key defined on the access route, a single value LOOKUP will be retrieved into the Predetermined Join Field. For example a product description can be retrieved using the product code if an access route has been defined between an order line file and a product file. A Keep Last value can be nominated to reduce I/Os.

When the relationship is 1:many (number of records retrieved is greater than 1) the Predetermined Join Field will hold the result of the selected operation on the selected field in the file nominated by the access route, retrieved using the key in the access route definition. For example the total of the line values from an order line file could be retrieved for display with the order header file.

The access route on which Predetermined Join Fields are defined includes in its definition two attributes which affect Predetermined Join Fields. They are the Keep Last value and the Derivation. Both of these attributes apply to all Predetermined Join Fields defined on the access route.

The Keep Last value only affects Predetermined Join Fields when the access route relationship is 1:1. It causes the last "nnn" values retrieved from the accessed file to be stored in memory and thereby reducing I/Os.

The Derivation attribute nominates when the Predetermined Join Fields are to be retrieved from the accessed file, i.e. either before or after virtual fields which are derived 'After input from file'. Predetermined join fields will be displayed when prompting for fields in the RDML editor in the same way as virtual fields.

Predetermined join fields are dependent on the access route they are defined on.

#### Warning

• Predetermined join fields are very efficient when used to look up descriptions in code files where the file accessed is a high speed table but overuse of this facility could cause the opposite effect, particularly in DBOPTIMISED functions when the accessed file is not a high speed table.

## As a guide, do not access more than 10 to 15 files for Predetermined Join Fields.

#### Also See

- 4.7.2 PJF Examples
- 4.7.3 PJF Example using Personnel System
- 4.7.6 PJF Considerations
- 1 4.7 Predetermined Join Field Development

## 4.7.2 PJF Examples

#### Example 1

To retrieve a customer description to be displayed when processing an order:

- First an access route is defined from the order file to the customer file. The maximum records retrieved is defined as One and the customer code field from the order file is used as the key.
- Next a Predetermined Join Field is added to receive the customer description using the LOOKUP operation. The source field name can be used and, if required, a KEEP LAST value entered.

When the I/O module for the order file is made operational the Predetermined Join Field customer description will be available in the same way as a virtual field when coding RDML without having to include any type of fetch from the customer file.

Where a code file is repeatedly accessed for the same information an appropriate KEEP LAST value will reduce the number of I/Os performed.

#### Example 2

To retrieve the descriptions of the 'from' and 'to' warehouses in a stock transfer file:

- Two access routes must be defined, as in Example 1, from the transfer file to the warehouse file. The first will use the 'from' warehouse code field as the key and the second will use the 'to' warehouse code field as key.
- When defining the Predetermined Join Field on the first access route, the source field name can be used; but when defining the Predetermined Join Field on the second access route, a different field name must be used so that both the 'from' and 'to' descriptions are available at the same time.

#### Example 3

To retrieve the total value and number of lines in an order when displaying the order header details:

- Define an access route from the order header file to the order line file with the appropriate key. The number of records expected will be the maximum number of lines in an order.
- Select the access route for Predetermined Join Field definition.
- From the displayed fields in the order line file, select the line value field. Add the Predetermined Join Field using the TOTAL operation. The count of

order lines can also be performed on this field or on any other field in the file.

When the order file is made operational, the total value and line count fields are available in the RDML editor along with the order fields without coding any reference to the order line file.

#### Example 4

To calculate a price for each line of an order using a Predetermined Join Field and a virtual field:

- Define an access route from the order line file to the product file with a 1:1 relationship (by setting the Maximum Records value), a Keep Last value if appropriate, set the Derivation to "before virtual fields" and use the product code field from the order line file as the key.
- Define a virtual field of extended definition type "mathematical calculation" to be derived "after input from file". Define the calculation as follows: product quantity (from order line file) multiplied by product price (Predetermined Join Field from product file) equals virtual field.

By including the product price (Predetermined Join Field) and product value (virtual field) when defining a select command in RDML, the arithmetic will be performed without any additional RDML code.

### Also See

4.7.1 Predetermined Join Field Concepts

## 4.7.3 PJF Example using Personnel System

Consider the Department and Section files which are part of the Personnel System Application database.

Imagine that you want to be able to make the Department Description field in the Department file part of the Section file. You also want to derive the Section Count field by counting the sections for a specific department in Section file. PJFs will be created to make these two fields part of the Department file.

First, two new virtual fields are created in the repository:

- Description of Department
- Section Count

Based on the access routes between the Section file and the Department file, a PJF is added to the Section file so that the Description of Department field is a file lookup to the Department Description in the Department File. Description of Department is a virtual field. When it is used by a function, the repository will actually access the Department file to get the required data since it does not really exist in the Section file. The access route is used to determine how to access the data.

Based on the access routes between the Department file and the Section file, a PJF is added to the Department file so that the Section Count field is calculated by counting the appropriate sections in the Section file. Section Count is a virtual field. When it is used by a function, the repository will actually access the Section file to get and total the data.

#### Also See

#### Personnel Demonstration Application

## 4.7.4 Steps to Create a PJF

Following is a brief summary of the steps to create a PJF:

- 1. Create file access routes. The file access routes should have already been defined as part of the file definition steps. If they have not yet been set up, you will be required to do so before beginning.
- 2. Add field for PJF to the repository. Like other virtual fields, the field used for the PJF must be created in the repository.
- 3. Set up access route characteristics. The access routes contain two characteristics specific to the PJF, "Keep Last" and "PJF before/after Virtual field derivation". These two values must be set.
- 4. Define PJF characteristics. Using the access routes, the fields are linked in the two files and the type of operations performed are specified.
- 5. Make file operational. Once the PJF has been added, the file must be made operational.

It is important not to confuse PJFs with virtual fields. PJFs will appear on the list of fields in the file but they are maintained via the file access routes.

## 4.7.5 Access Routes and PJFs

There are some important PJF characteristics that are defined as part of the access route definition:

- Maximum Records
- Keep Last
- Derivation Before or After Virtual Field.

Both of these attributes apply to all predetermined join fields defined on the access route.

#### **Maximum Records:**

This value specifies the maximum number of records that are expected to be found in the Accessed File that have a key matching the Key Fields/Values specified. When the value is One then a "1:1" relationship between the files is established. If the value is More than one, then a "1:many" relationship is established. A 1:1 relationship will allow a file LOOKUP type PJF to be created and the Keep Last value can be entered. A 1:m relationship will allow TOTAL, MAXIMUM, MINIMUM, AVERAGE and COUNT type PJFs to be created.

#### **Keep Last:**

When the file relationship is 1:1 using the key defined on the access route, a single value will be retrieved into the predetermined join field. In the Personnel System example, a Department Description can be retrieved using the Department Code if an access route has been defined between the Section file and Department file. A KEEP LAST value can be nominated to reduce I/O operations. Keep last causes the last 'nnn' values retrieved from the file to be stored in memory. If a value with the same Department Code is required, it can be read from memory instead of reading from the file again.

#### **Derivation Before/After Virtual Field:**

The derivation of the PJF before or after virtual fields will affect the retrieval timing. This option nominates when the predetermined join fields are to be retrieved from the file, i.e. either before or after virtual fields which are derived. "Before" allows a predetermined join field to be used in deriving a virtual field. "After" allows a virtual field to be used as a key field to a file being accessed by the predetermined join field. Simply ask yourself, do I need the virtual field to determine the PJF? If yes, specify After. Or ask yourself, do I need the PJF to determine the virtual field? If yes, specify Before.

## 4.7.6 PJF Considerations

Following are some important considerations for using PJFs:

- Although the predetermined join field processing is coded within the OAM, the related files are only accessed when one of the predetermined join fields is specifically used. This make PJFs very efficient, i.e. they do not impact file access times when they are not used.
- If the file to be accessed by the access route is defined as an IBM i high speed table, the source information for the predetermined join fields will be retrieved from the high speed table, thereby improving performance and reducing I/Os.
- When defining predetermined join fields, careful thought should be given to their use. They are very efficient for looking up code files for descriptions when used with a KEEP LAST value or for accessing a high speed table. If they are used excessively, each file accessed will affect the number of files used in the OAM and DBOPTIMISED functions. (For more details, refer to Using \*DBOPTIMIZE / \*DBOPTIMIZE\_Batch in the LANSA for i User Guide.)
- If the function has been DBOPTIMISED the predetermined join field processing will have been included in the OAM logic which is embedded in the function.
- PJFs could greatly increase the number of I/Os performed by the OAM where there are 1:m relationships. For example, each time a single record is read from the Department file, it must access all related records in the Section file if the Section Count field is required.
- Predetermined join fields will appear in the list of fields in the file, but they are maintained using the access routes.

# As a guide, do not access more than 10 to 15 files for Predetermined Join Fields.

### 4.8 Batch Control Development

A feature of the LANSA Repository, batch control can be used to maintain numeric totals of fields in different files. To understand how to use batch control, review the following topics:

- 4.8.1 Batch Control Concepts
- 4.8.2 Batch Control Examples
- 4.8.3 Steps for Defining Batch Control
- 4.8.4 Batch Control Considerations

#### Also See

Batch Control Tab in the User Guide

Batch Control Definition in the Technical Reference Guide

1 4. Creating Application Databases

## 4.8.1 Batch Control Concepts

Batch control is a special feature of the LANSA Repository. Batch control logic is used to define the logic by which a numeric field in one file is totaled into the field of another file. LANSA's batch control logic automatically maintains the control totals in files for you as records are inserted, updated and deleted. (The fields used in batch control can include key fields.)

Batch control is often used as a means of verifying totals during data entry. For example, batch control could be used to maintain an order header file with a field which stores the total value of a given order. This field is calculated by adding up the order amounts from each of the order lines for the specified order. Batch control could also be used to keep track of the total number of lines in a given order.

Batch control logic in files can improve performance by reducing the amount of database I/O. It also simplifies application logic by centralizing calculations and performing them automatically. Batch control logic is defined at the file level. LANSA implements batch control using its object access modules (OAM).

Functions equivalent to batch control can be performed using file level triggers. Though the trigger approach requires manual coding of the trigger functions, the ability to manually code the application logic makes the trigger very flexible and very powerful. Triggers are the recommended approach for implementing batch control type logic.

Important: Please note that the use of batch control logic, and the repository fields used to support it, may not be consistent with the practices of logical data modeling.

Following is an example of batch control:

• A "batched transaction" data entry application may use three control files structured like this:

ata File
CTION
NT

	DESCRIPTION
	DEBIT
	CREDIT

- In this structure, BDEBIT and BCREDIT are totals of all DEBIT and CREDIT values for a particular DATE and BATCH.
- Likewise, DDEBIT and DCREDIT are totals of CREDIT and DEBIT values for a particular DATE.

When setting up the definition of the "Entry data" file in this example, the batch control logic component would be used to indicate that:

- Fields DEBIT and CREDIT in the "Entry data" file should be accumulated by DATE and BATCH into fields BDEBIT and BCREDIT in the "Batch totals" file.
- Fields DEBIT and CREDIT in the "Entry data" file should be accumulated by DATE into fields DDEBIT and DCREDIT in the "Daily totals" file.
- When the I/O module for the "Entry data" file is compiled (see beginning of this chapter for discussion of I/O modules) it will contain logic to automatically maintain the batch control fields in the "Batch totals" and "Daily totals" files.
- This means that whenever a record is added, updated or deleted from the "Entry data" file the totals will be maintained. No other logic is required to maintain the totals.
- The logic generated will handle changes to the batch control "keys" as well. Thus even if a record in the "Entry data" file was changed from BATCH 1234 to 4567 and DATE 010187 to 150187 the control totals will be maintained correctly.
- If a user creates a function to add records to the "Entry data" file it will use the I/O module. The I/O module will maintain the batch control totals in the other 2 files without you even knowing of their existence.

Following is another, simpler example. An order header file and an order lines file have the following fields defined in them:

#### Order Header File Order Lines File

ORDNUM	ORDNUM
CUSTNO	ORDLIN
DATE	PRODNO
TOTDUE	QUANTY
	VALUE

• In this structure, TOTDUE (in order header) is a count of all associated VALUE fields in the order lines file. TOTDUE is the total value of all associated lines in the order.

#### Also See

4.8.2 Batch Control Examples

1 4.8 Batch Control Development

## 4.8.2 Batch Control Examples

These examples apply to defining batch control logic within a file definition.

#### Example 1

The "batched transaction" data entry application example used at the beginning of this section has 3 control files that were defined as follows:

File name:		DAYTOT	BCHTOT	ENTDTA
Description:		Daily totals	Batch totals	Entry data
Fields:		DATE	DATE	DATE
		DDEBIT	BATCH	BATCH
		DCREDIT	BDEBIT	TRANSACTION
			BCREDIT	ACCOUNT
				DESCRIPTION
				DEBIT
				CREDIT
Primary key	1	DATE	DATE	DATE
	2		BATCH	BATCH
	3			TRANSACTION

In this structure BDEBIT and BCREDIT are totals of all DEBIT and CREDIT values for a particular DATE and BATCH. Likewise, DDEBIT and DCREDIT are totals of CREDIT and DEBIT values for a particular DATE.

To define this particular batch control logic requirement to LANSA the following batch control logic details should be added to the definition of file **ENTDTA (Entry data):** 

DefinitionBatch CarNameFile Nam	tl Linked Fields E In This File		5
---------------------------------	------------------------------------	--	---

				File
ENTDTA	BCHTOT	DEBIT	BDEBIT	DATE
		CREDIT	BCREDIT	BATCH
ENTDTA	DAYTOT	DEBIT	DDEBIT	DATE
		CREDIT	DCREDIT	

#### Example 2

The second example at the beginning of this section dealt with an order header and order lines file that were defined as follows:

File name:		ORDHDR	ORDLIN
Description:		Order header	Order lines
Fields:		ORDNUM	ORDNUM
		CUSTNO	ORDLIN
		DATE	PRODNO
		TOTDUE	QUANTY
			VALUE
Primary key	1	ORDNUM	ORDNUM
	2		ORDLIN
	3		

In this structure TOTDUE (in order header) is a count of all associated VALUE fields in the order lines file. i.e. TOTDUE is the total value of all associated

invoice lines.

To define this particular batch control logic requirement to LANSA the following batch control logic details should be added to the definition of file **ORDLIN (order lines):** 

File Definition Name:	ORDLIN
Batch Control File Name:	ORDHDR
Linked Fields In this File:	VALUE
Linked Fields In Control File:	TOTDUE
Keys to Access Control File:	ORDNUM

1 4.8 Batch Control Development

## 4.8.3 Steps for Defining Batch Control

To create a batch control definition, you must specify four pieces of information:

- 1. The file which will contain the batch control total
- 2. The field in this file which is being totaled
- 3. The field in the control file which will contain the batch control total
- 4. The keys used to access the control file, i.e. the fields in this file needed in order to access the control file.

Up to four pairs of fields can be defined for the batch control definition.

## 4.8.4 Batch Control Considerations

Once batch control is created between two files, it will be necessary to delete the batch control definition before attempting to delete either of the files being used.

It is important to note that batch control and file triggers work with the real fields in the file, whereas PJFs use virtual fields. Depending on the database design, one technique may have advantages over another.

Equivalent functionality to batch control can be performed using file level triggers. Though the trigger approach requires manual coding of the trigger functions, the ability to manually code the application logic makes the trigger very flexible and very powerful. Triggers are the recommended approach for implementing batch control type logic.

Predetermined join fields can also be used for calculating totals between files. When deciding if a PJF should be used instead of batch control, it is very important to note the difference in these two techniques. Batch control keeps a total in an existing real field in a file. The value is maintained when the dependent file is updated. With the PJF, the field is a virtual field. It is not part of the file. The PJF total is calculated when the field is accessed in the file.

Batch control is fully supported with RDML files. However, when used with RDMLX files, it will work only with RDML keys and target/source fields.

1 4.8 Batch Control Development

## **4.9 Compiling Files**

To understand how to make a file operational and when to compile files in your database, review the following topics:

- 4.9.1 Making a File Operational
- 4.9.2 What about Data?
- 4.9.3 What to Recompile
- 4.9.4 When To Recompile

#### Also See

File Compile Options in the Technical Reference Guide

1 4. Creating Application Databases

## 4.9.1 Making a File Operational

In LANSA, making a file operational may involve three different tasks in the following order:

- creating a physical file
- creating logical files over a physical file
- creating an OAM (Object Access Module or I/O module)

When a file is first defined, it must be made operational and all three tasks are to be performed, (assuming there are logical files), by either you or LANSA.

If you change your field definitions or your file definitions in the repository, you may need to perform some or all of the above tasks as you make the files operational once more. All three tasks do not always have to be performed. For example, adding a validation check will not require recompiling the physical or logical files. (Refer to 4.9.3 What to Recompile.)

In most circumstances, LANSA will check which of the three tasks needs to be performed; however, LANSA cannot detect all changes made at the field level. For example, if you change a default value for a field on a file, you should use impact analysis and then tell LANSA to recompile the OAM when you make the file operational. If you changed a field's length, LANSA will know to perform all three tasks of rebuilding files, logical views and OAM. (Refer to 4.9.4 When To Recompile.)

Making files operational is an excellent example of how LANSA shields developers from the underlying technical layers of the database and operating system. Under IBM i, making files operational will create physical and logical files. Under Windows, making files operational will create SQL tables in the appropriate database manager.

To make a file operational, use the Compile File option. The Compile Option dialog will appear. This dialog allows you to specify which tasks should be performed when the file is made operational.

Using the Compile File only if Necessary option, LANSA will check if the file compile needs to be performed; however, LANSA cannot detect all changes made at the field level. For example, if you change a field level rule for a field in the file, you must tell LANSA to recompile the OAM when you make the file operational. If you changed a field's length, LANSA will know to recompile the file.

When a file is recompiled, you have the option to save and reload and existing

data in the table. The types of changes to your table will determine how much of the data can be restored. (Refer to 4.9.2 What about Data?.)

You do not need to keep the generated source code unless you are planning to move your application to another platform which requires the C code (for example, Linux).

#### Also See

File Compile Options in the Technical Reference Guide

1 4.9 Compiling Files

## 4.9.2 What about Data?

What happens if you have a file filled with data and then you make a change to the file? When you make a file operational, you can specify save and restore data options. LANSA will attempt to copy the data to the new file where possible.

You may use the Save Table Data option when you wish to have the existing data in the table(s) saved before the table is deleted and recreated during a LANSA defined file generation/creation. Using this option will result in all existing data being saved to a file named ffffffff.DAT (where fffffffff is the table name) in the partition directory X\_LANSA\_ppp (where ppp is the partition identifier) in delimited ASCII format.

Using the Reload Table Data option will attempt to restore data back into the table using the data saved in the ffffffff.DAT file. It may not be possible to restore data in all circumstances. For example, if a field in the file is changed from alphanumeric to packed, it will not be possible to copy the data between fields.

If you have a considerable amount of data and you make major changes to the file, you may wish to write your own data reload functions. In this case, choose only the Save table data option and then execute your own reload function once the file compile is complete.

#### Also See

File Compile Options in the *Technical Reference Guide* ↑ 4.9 Compiling Files

## 4.9.3 What to Recompile

The following example will help you to understand what to recompile when a change occurs in your application database.

Consider the following:

Each file has its own OAM.

The DEPTAB file has an OAM to allow functions to update records from DEPTAB file.

The PSLMST file has an OAM to allow functions to update records from PSLMST file.

The PSLMST OAM may have rules which read from the DEPTAB file.

When you change the physical file definition of the DEPTAB File (i.e. add fields, remove fields, or change their attributes), the DEPTAB file, its logical files and its OAM must be created. The question is, "Does the PSLMST OAM have to be created?"

The PSLMST file has a validation check which involves the DEPTAB file. The Department Code in the Employee file must exist in a DEPTAB file when creating a new Employee. The PSLMST file's OAM must open the DEPTAB file to perform a file look up validation check because the Department Code exists in the DEPTAB file. The check is not done by calling the DEPTAB file's OAM. Therefore, the PSLMST file's OAM must also be recompiled or else an IBM i level check error will result. (For more information on level checks, see the IBM Database Guidelines.)

As a rule:

If you change a physical file definition, you must make it operational. You must also recompile any OAM for files which access the changed physical file.

Reviewing your data model relationships will help you find any affected files. You must also be aware of any validation rules you have added involving file look ups.

If you have used batch control, you will also be required to recompile OAMs.

#### Also See

4.9.4 When To Recompile

↑ 4.9 Compiling Files

# 4.9.4 When To Recompile

This chart summarizes when you will be required to make the physical file, the logical file and the OAM operational after making a specific change.

Trigger functions are included in the chart because they might need to be recompiled more frequently than 'normal' functions. The file level trigger functions have a list of the fields in the file, in the correct order, including virtual fields. If anything like field type, length, order of fields, etc. changes, the trigger must be recompiled.

Trigger functions and normal functions may or may not require a recompile, depending on how the functions use the changed fields. This is indicated with 'O' for Optional.

**Note:** A trigger and a trigger function are not the same thing. A trigger is the definition of when a function will be called. It consists of trigger points, trigger conditions and the name of the trigger function. A trigger function is an RDML coded program that defines the action to be taken.

Type of Change	Physical File	Logical File	OAM	Trigger Function	Function	Detected
Field Length/Type	Y	Y	Y	Y	0	Y
Column Heading/Label	Ν	Ν	Ν	0	0	-
Edit code/word	Ν	Ν	Ν	0	0	-
Default Value	Ν	Ν	Y	0	0	Ν
Output Attribute	Ν	Ν	Y	0	0	Ν
Display Attribute	Ν	Ν	Ν	0	0	-
Prompt	Ν	Ν	Ν	Ν	Ν	-
Help Text	Ν	Ν	Ν	Ν	Ν	-
Authority	Ν	Ν	Ν	Ν	Ν	-

Validation Rules	Ν	Ν	Y	Ν	Ν	Y
Virtual Fields	Ν	Ν	Y	0	0	Y
Batch Control	Ν	Ν	Y	Ν	Ν	Y
Access Routes	Ν	Ν	Ν	Ν	Ν	Y
Logical File	Ν	Y	Y	Ν	Ν	Y
Physical Files	Y	Y	Y	0	Ν	Y
Trigger Point	Ν	Ν	Y	0	Ν	Y
Trigger Condition	Ν	Ν	Y	Ν	Ν	Y

Refer to the LANSA Web site technical support pages (www.lansa.com/support) for more details.

↑ 4.9 Compiling Files

# 4.10 Load Other File

Visual LANSA allows you to load existing IBM i or SQL tables into the LANSA Repository in order to create Other File definitions. These files can then be accessed through LANSA in a similar manner to LANSA created files, however, the file definitions cannot be altered because they are Other files.

To understand how to load and use Other files, review the following topics:

- 4.10.1 Other File Concepts
- 4.10.2 PC Other Files
- 4.10.3 IBM i RDMLX Other Files
- 4.10.4 Setting Other File Load Options

#### Also See

## 4.10.1 Other File Concepts

Load Other File creates a file definition with LANSA fields representing the columns in the table. The fields may be existing fields or they may be created when the file definition is loaded. Existing fields are only used if there is an exact match of the field name, data type, length, number of decimals, etc. Any variation from existing fields will cause a new field to be created.

Some native database types are not supported by LANSA, and Load Other File will automatically skip loading columns of these types. If you will be inserting data into the file via LANSA applications, you must ensure that these skipped columns have a default at the database level.

If the primary key contains a skipped column, an IBM i Other File will usually be defined in LANSA with no primary key. For PC Other Files, if another unique index exists, this may be loaded as the primary key and may also be loaded as a logical view.

If a logical file has a skipped column as one of its keys, the logical file will not be loaded.

When loading an other file in an RDMLX-enabled partition, the full range of field types supported by LANSA is available, regardless of the Available field types in the partition maintenance dialog. This allows you to automatically use the full features of LANSA to support Other Files, without having to be concerned about developers using new field types in other applications.

Occasionally, you may find that Load Other File creates a field that is not the type or length you expected, or perhaps you expected it to reuse an existing field. The solution is to add a virtual field to the file that converts the loaded field to what you expected. These fields will be preserved if the file is reloaded.

When loading DateTime fields in an RDMLX-enabled partition, Load Other File assumes that the field will be stored and displayed in the local time zone. If this is not the case, add the SUTC and/or DUTC attributes to the field. For further details, refer to Datetime in the *Technical Reference Guide*.

After loading Other files, you may change keyboard shift settings for some Other File fields. To do this, just open them in the LANSA Editor, go to the Field Definition tab and select the suitable keyboard shift values for them.

LANSA cannot make use of an Other File that contains only fields of type BLOB or CLOB, as it requires other fields to be used to query the BLOB or CLOB fields.

When loading other files, LANSA will recognise an Identity (Autonumber) column if one exists on the file. Refer to Identity columns.

There are many database and table-specific reasons why a column or table cannot be inserted or updated. If you attempt to update or insert a row into an Other file, and an error not related to LANSA is returned, you need to investigate the cause. Validations that the database may perform are not checked by LANSA and so the database may reject the data with a fatal error. Rules should be added to LANSA to perform these checks. If the cause cannot be avoided from within LANSA, you will only be able to read the table. For PC Other Files, you may delete a problematic field from the file definition, or mark the file as allowing read access only.

#### Also See

4.10.2 PC Other Files
4.10.3 IBM i RDMLX Other Files
4.11 LANSA Internal Tables
↑ 4.10 Load Other File

# **Identity columns**

There may only be one Identity column per file. If the table being loaded actually has multiple identity columns, Load Other File will only count the first identity column loaded as an Identity column

Identity columns cannot be set to a specific value on insert or update. If an identity column is one of the primary key fields, it must be specified in the FIELDS(...) parameter of the INSERT command. However, the value in the field will not be inserted into the table, and the value of the field after the INSERT will not reflect the value that was actually inserted.

They are automatically generated on insert. Note: there is no way to retrieve the value without re-reading the row that was just inserted. To re-read the row, you will need to ensure the file has a primary key or unique logical file or index that does not contain the identity column.

**Note:** When compiling IBM i Other Files on platforms other than IBM i, if you choose to drop the table and reload the data, identity column values will be regenerated.

#### Also See

4.10.1 Other File ConceptsCreate RRNO Column↑ 4.10.1 Other File Concepts

# Handling of Timestamp or DateTime columns

These notes apply to the datatypes recognised by ODBC as SQL\_TIMESTAMP, as well as the IBM i native datatype Timestamp (Z). These datatypes contain a date and time in one column.

When a timestamp column is loaded into the LANSA Data Dictionary as a field without the ASQN attribute, the default value '1900-01-01' is assigned. This value must be reviewed to verify its suitability.

The length of the field is between 19 and 29, depending on the native datatype.

The system variables \*TIMESTAMP\_DFT, \*TIMESTAMP\_LOVAL, and \*TIMESTAMP\_HIVAL are also available. On platforms other than IBM i, \*TIMESTAMP\_DFT and \*TIMESTAMP\_LOVAL both evaluate to '1900-01-01 00:00:00.000000'. On IBM i, these system variables evaluation to '0001-01-01.00.00.000000'. On both platforms, \*TIMESTAMP\_HIVAL evaluates to midnight on 9999-12-31.

The \*TIMESTAMP\_XXX variables may only be specified as the field's default value where the field length is 26 or higher. If you want to use these values with a shorter field, use a virtual field of length 26.

If the field is of type Alphanumeric, one of the following formats must be used:

• Full timestamp (YYYY-MM-DD HH:MM:SS[.f....] Seconds must always be specified, and the fractional part may only be specified where the native datatype supports it, and only for the maximum length the field was set to when added to the Data Dictionary.

After a fetch from the file, the field will always be in this format, unless it is NULL, in which case it will be blanks. If the row is then updated, LANSA will automatically set the column to \*TIMESTAMP\_DFT as blank is not valid.

- Date only (YYYY-MM-DD) If the field is inserted/updated, the time portion will be automatically set to 00:00:00
- Time only (HH:MM:SS.[f...]) If the field is inserted, the date portion will be automatically set to '1900-01-01'. The same format rules apply as for the full timestamp. Do not specify a fractional part if it cannot be specified in the full timestamp format.
- If the field is used in an I/O command and is not in a valid format, a fatal

error will occur. On insert or update some datatypes may be successfully converted by the DBMS from other formats, although this is not recommended. If data may be manually entered, you should use virtual fields to convert to and from the above formats.

#### Also See

Using SQL Server datatype Smalldatetime4.10.3 IBM i RDMLX Other Files↑ ↑ 4.10.1 Other File Concepts

# How BLOB and CLOB columns are handled by RDMLX Other Files

For IBM i Other Files, refer to No additional LOB columns.

For PC Other Files:

- if you select *Add columns to support LOBs* refer to Additional LOB columns
- If you did NOT select *Add columns to support LOBs* refer to No additional LOB columns

Also See

RDMLX PC Other File ↑ 4.10.1 Other File Concepts

# No additional LOB columns

This applies to all IBM i Other Files and PC Other Files when *Add columns to support LOBs* has NOT been selected.

When reading rows back from the file, if the BLOB or CLOB is not NULL, a random filename will be generated. The filename will have the extension .txt for a CLOB, and .bin for a BLOB, and will be unique in its directory.

Additional LOB columns

**RDMLX PC Other File** 

 $\uparrow \uparrow 4.10.1$  Other File Concepts

# 4.10.2 PC Other Files

LANSA supports most popular databases used in business at stable and supported versions. Refer to your LANSA supplier if you need exact details of what databases and versions are supported.

A PC Other file will always be loaded into an RDMLX partition as an RDMLX PC Other File. There is no option to load a PC Other file into an RDMLX partition as an RDML file. Furthermore, RDML field types will be converted to RDMLX field types. For example, alphanumeric fields become string fields. When LANSA loads the file, it records the information required to connect to the database, which is then automatically used at execution time. These details may need to be altered. (Refer to Using DEFINE\_DB\_SERVER to modify connection details.)

- If LANSA cannot find a primary key defined for the file, it will load the first unique index (if any) it finds as the primary key.
- The Create RRNO Column flag must be set as appropriate for your application. Refer to Create RRNO Column.
- DateTime fields may require special handling. (Refer to Handling of Timestamp or DateTime columns.)
- It is strongly recommended that you do not load LANSA files as PC Other Files. Refer to LANSA Files loaded as PC Other Files.
- SuperServer PSXX parameters cannot be used with local PC Other Files. (Refer to Using SuperServer with PC Other Files.)

PC Other File data cannot be shipped with the Deployment Tool. If you need to ship data, you will need to write your own functions to insert or update the data as part of a package install. The Built-In Functions TRANSFORM\_LIST and TRANSFORM\_FILE may be appropriate. For further details, refer to Commands to Execute in the LANSA Application Deployment Tool Guide.

#### Also See

LANSA Files loaded as PC Other Files Using SuperServer with PC Other Files Using DEFINE\_DB\_SERVER to modify connection details Database-specific Notes RDML PC Other File RDMLX PC Other File Load Other File in the *Technical Reference Guide* ↑ 4.10 Load Other File

# **RDML PC Other File**

When loading a PC Other File in a partition that is NOT enabled for RDMLX, LANSA supports a very limited range of database columns. It is highly recommended that you enable your partition for RDMLX when use of PC Other Files is required.

RDML Other Files do not recognize SQL NULLs. (Refer to Handling of NULLs.)

RDML Other Files support a maximum Alphanumeric field length of 256. (Refer to Handling of columns longer than 256 bytes.)

RDML Other Files support numeric fields with a maximum scale of 30 and a maximum precision of 9, and floating point numbers are loaded as fixed point numbers. (Refer to Handling of large numeric columns.)

#### Also See

**RDMLX PC Other File** 

↑ 4.10.2 PC Other Files

# Handling of NULLs

When a row is fetched from the database, LANSA replaces NULL values with 0 (zero) or blanks, depending on the field type. If the row is then updated, LANSA will overwrite the NULL in the database with the zero or blank value (unless you have changed it). The only exception to this is that if a timestamp field is blank, it will be set to 1900-01-01 (the default value for timestamp fields) as blank is not valid.

Do not use a column containing a NULL value in a where condition or as part of a key. LANSA will search for values matching zero or blanks and will not find values that are NULL. In fact, if the field is a timestamp, a fatal error will occur as blank is not accepted by ODBC/SQL as a comparison value for a timestamp.

Do not use a column that contains a NULL value as a key to update a row. If you do, LANSA sets it to zero or blanks and does not retain the NULL value. As a result, a match will not be found when the update is attempted and LANSA will not perform the update.

↑ RDML PC Other File

# Handling of columns longer than 256 bytes

Columns longer than 256 bytes will be created as 256 byte Alphanumeric fields. SELECT and FETCH will only be able to access the first 256 bytes (if there are more than 256 bytes, a warning message will be usually issued). This means that UPDATEs of rows containing such a column will be TRUNCATED to 256. This may result in data being lost that is updated by an external application.

It is highly recommended that you use these tables for reading only. Alternatively, you may delete the field from the file.

**RDML PC Other File** 

# Handling of large numeric columns

Columns with a precision higher than 30 will be created as a field with a length of 30. Columns with a scale higher than 9 will be created with decimals of 9.

Floating point columns are loaded as Signed(30,9). If this is not suitable, use a virtual field that is more appropriate.

Values with more than 30 significant digits or 9 decimal places are not handled. Data may be truncated at either end.

If you know that the values are unlikely to exceed the field length and decimals, you may create a virtual field of (30,9) that is set to the value of this column. Alternatively, you may delete the field from the file.

**RDML PC Other File** 

## **RDMLX PC Other File**

When loading a PC Other File in an RDMLX partition, LANSA supports all database columns recognized by ODBC, except for type SQL\_INTERVAL. For details on the various field types, refer to Field Types in the LANSA Technical *Reference Guide*.

At the time of loading, you can choose whether to Add Columns to Support LOBs. (Refer to How BLOB and CLOB columns are handled by RDMLX Other Files.)

Also See

RDML PC Other File ↑ 4.10.2 PC Other Files

# Additional LOB columns

This applies to PC Other Files when *Add columns to support LOBs* has been selected.

At file build time, additional columns will be added to your table to allow the storage of BLOB and CLOB as per LANSA files. For each BLOB or CLOB field, a filename column will be added, with the initial value of NULL. When an Insert or Update of a file is carried out by LANSA, a filename value will be stored into this column.

When reading rows back from the file, if the BLOB or CLOB is not NULL, but the filename column is NULL (as LANSA did not do the insert, and has not updated the row), a random filename will be generated. The filename will have the extension .txt for a CLOB, and .bin for a BLOB, and will be unique in its directory.

If the filename column has a value at read time, that filename will be used when creating the file. This is a useful feature when you want to know what the filename will be at read time.

#### Also See

# **Create RRNO Column**

This option specifies whether or not the X\_UPID and X\_RRNO columns should be added to the table when the file is built. If *Create RRNO Column* is selected, any existing identity column will be used instead of X\_RRNO, as a second identity column cannot be added to the table. (Refer to Identity columns.)

If you load a file that does not have a primary key, and do not select *Create RRNO Column*, functionality will be limited. For further details, refer to Create RRNO Column in the *LANSA Technical Reference Guide*. To minimize functionality limitations, add a primary key or unique index and reload the table. Alternatively, select the *Create RRNO Column* option in the *File Attributes* for the file.

The X\_RRNO value will be automatically incremented using a database-specific method.

You must be connected to the database with appropriate permissions on the table to allow the column(s) to be added.

Once the column(s) have been added to the table, they can only be removed manually.

Note that adding the RRNO column affects existing SQL syntax which does not explicitly specify column names. For example the INSERT statement:

INSERT TO MYTABLE VALUES('A', 'B', 2)

only works if a value is specified for all columns. After you add the RRNO column this syntax will not work anymore.

It is better programming practice to specify the names of the columns. In this way adding new columns to the table will not affect your code:

INSERT TO MYTABLE(COL1, COL2, COL3) VALUES('A', 'B', 2)

#### Also See

LANSA Files loaded as PC Other Files

4.10.3 IBM i RDMLX Other Files

Building a File with Create RRNO Column selected (Oracle-specific)

Removing X\_UPID and X\_RRNO Columns (Oracle-specific)

↑ 4.10.2 PC Other Files

# LANSA Files loaded as PC Other Files

A better way to access LANSA files, than loading them as PC Other files, is to import the file definition into the current partition (and use DEFINE\_DB\_SERVER to redirect the RRNO file, if Auto RRN is not specified). Only load LANSA Files as Other Files when its impossible to achieve the desired outcome by importing the File Definition.

*Create RRNO column* must not be specified for tables that are actually LANSA Files and for which the original LANSA file definition that created the file has the option *Create RRNO column* selected.

If the original LANSA file definition has Create RRNO column specified and not Auto RRN then the solution is to deselect the Create RRNO Column option in the file attributes (after loading the file) and use the BIF

DEFINE\_DB\_SERVER to redirect the RRNO file to the originating LANSA system's partition directory.

If the original LANSA file definition has Create RRNO column specified and Auto RRN specified, it will function just like any other LANSA file created directly by this partition.

#### Also See

DEFINE\_DB\_SERVER in the LANSA Technical Reference Guide ↑ 4.10.2 PC Other Files

## **Using SuperServer with PC Other Files**

If you want to use PC Other Files on the client, you cannot use the PSXX parameters to connect to the primary server, as this connects all files (\*) to the server. Instead

- Use DEFINE\_OS\_400\_SERVER or DEFINE\_ANY\_SERVER to define the connection to the primary server.
- Use CONNECT\_FILE to nominate specific files or wildcards to be connected to the primary server, ensuring that the PC Other Files are not specified.
- Use CONNECT\_SERVER to connect to the server.

Note that the PSXX parameters can be used if PC Other Files are on the server rather than the client.

↑ 4.10.2 PC Other Files

# Using DEFINE\_DB\_SERVER to modify connection details

When LANSA connects to the database it records the information required to connect to the database. This is used at execution time to connect to the database. The most important information is the ODBC DSN, user id and password. There are many other parameters, most of them specific to a particular database. Please review the documentation that comes with your database for more information on these other parameters.

The ODBC DSN, user id and password may need to be changed in the executable environment. For example you may want to prompt for the userid and password every time the user connects, or use different DSNs for development, test and production systems. This can all be achieved using the DEFINE\_DB\_SERVER BIF, and optionally the CONNECT\_SERVER BIF.

#### Also See

Microsoft Access Database ODBC data source

↑ 4.10.2 PC Other Files

## **Database-specific Notes**

# **Microsoft SQL Server**

## Tables that do not belong to owner DBO

In previous versions of LANSA, the table owner was not used to access SQL Server Tables at execution time. This was because versions prior to SQL Server Version 2000 did not implement a complete owner solution.

For backward compatibility, existing configurations and compiled OAMs will continue with the original behavior, which is described in Original Behavior below.

If you use SQL Server as your LANSA database and you created it using a version of LANSA prior to LANSA V11.3, and if you have deployed applications that also use SQL Server, you must NOT use this new feature. You must continue to use a LANSA database created by prior versions of LANSA. This is because there is no migration possible from a database that does not support collections to one that does.

If you are using a SQL Server LANSA database for the first time with LANSA V11.3 or you use Oracle or Adaptive Server Anywhere as your LANSA database, then this feature can be used.

Previously, the generation of OAMs for SQL Server PC Other File would always leave the table owner out of the generated SQL. Now, the table owner can be included in SQL in the OAM if the following instructions are followed:

- **If you use SQL Server as your LANSA database**, install from the LANSA V11.3 CD (or later) into a new configuration, and rebuild the OAM for the file.
- **If you do NOT use SQL Server as your LANSA database**, modify X\_DBMENV.DAT. To do so, select the *Database* sub-menu item from your LANSA system's *Settings and Administration* menu.



Locate the settings for the SQL Server database, and ensure that these parameters are set as shown here: DATABASE\_TYPE=MSSQLS, SUPPORTS\_COLLECTIONS=YES. Rebuild the OAM for the file.

#### Note:

- No changes are required to deployed systems, only the generation of OAMs is affected.
- File does not need to be reloaded.
- New configurations running with SQL Server always use collections, so a table accessed at execution time must have the same owner as at load time.

# **Original Behavior**

If the table does not belong to dbo (Microsoft and LANSA both recommend that all tables belong to dbo), the result may be that the table may not be found at runtime, or a different table (with the same name) may be accessed instead.

Any SQL Server user may automatically create tables as dbo using the stored procedure sp\_addalias. For example: sp\_addalias user1, dbo. Refer to your SQL Server documentation for more details.

# Using SQL Server datatype Smalldatetime

SQL Server's Smalldatetime supports a very limited range of dates, therefore do not attempt to use the system variable \*TIMESTAMP\_HIVAL with fields of this type.

## Also See

# Handling of Timestamp or DateTime columns 4.10.2 PC Other Files

↑ Database-specific Notes

## **SQL Server Express Limitations**

SQL Server Express has limitations when compared with other editions of SQL Server. These limitations may mean that SQL Server Express will not meet the requirements of your particular implementation of LANSA.

## **General Limitations**

Maximum number of processors per instance limited to the lesser of 1 processor or 4 cores.

Maximum number of instances (AKA databases) 16

Maximum RAM	1 GB
Maximum database size	4 GB
SQL Server Agent	No

Given the RAM limitation, using just 1 CPU and depending on the amount of activity and speed of the machine, ten LANSA developers maybe an appropriate maximum for SQL Server Express. It's difficult to provide accurate guidance due to the many variables involved so it's important to test your particular needs before deciding which SQL Server Edition is appropriate for your development environment. LANSA uses the database heavily as it is a repository-based development system so it is well worthwhile investing in at least the Workgroup Edition when there are five developers or more running the database engine on a multi-processor high-end server with plenty of RAM. Whether you opt for Express Edition or something better, if more than one user is going to use the database, put the database engine on a separate machine using a server operating system.

SQL Server Express is compatible with the full edition, so upgrading to the Standard Edition or Enterprise Edition later is supported by Microsoft with some provisos as shown here: Version and Edition Upgrades on the SQL Server Developer Center. If you find performance is not as good as you might expect, LANSA would suggest that you improve the hardware and/or upgrade to a full edition.

The choice for deployed systems is more straightforward. 4 GB is not a very large database size and so Express Edition is unlikely to be sufficient. And if

that's large enough, the number of users that can be effectively supported with 1 GB RAM will probably be too few for most deployed systems. So, LANSA recommends not using Express edition for deployed systems.

Following are links for further information:

- A list of the major features of SQL Server Express and major omissions: SQL Server Express Features.
- A feature-by-feature comparison of the various editions of SQL Server: Features Supported by the Editions of SQL Server.

↑ Database-specific Notes

# **Microsoft Access**

# **Default value for X\_UPID may not be set automatically**

If the X\_UPID column is added to an MS Access table by LANSA (when *Create RRNO column* is selected) and if the table will have inserts made by a non-LANSA program, you must manually ensure that a default value is set for X\_UPID.

This is due to a limitation in some versions of MS Access that do not support the DEFAULT syntax in the ALTER TABLE statement. If this is not done and X\_UPID is NULL, when LANSA attempts to update a row inserted by the non-LANSA program LANSA will always report a crossed-update validation error.

# Microsoft Access Database ODBC data source

The default ODBC data source created by a Microsoft Access install is not suitable for use by Other Files, as it does not provide all connection parameters for the chosen database. Thus, when you come to use your Microsoft Access Other File, LANSA will supply to Microsoft Access an incomplete list of connection parameters, causing Microsoft Access to ask for the missing parameters. Every time you execute the application again, the Microsoft Access logon details will be asked for again.

There are two ways to address this issue. Firstly you can change the ODBC DSN to specify the missing information. This is the simplest and the most commonly used method. The second method is described in Using DEFINE\_DB\_SERVER to modify connection details.

## **SELECT command restrictions**

LANSA does not support Generic(\*Yes) or Options(\*StartKey). If you attempt to use these options on a SELECT command against an Microsoft Access Other File, you may get unexpected behaviour.

#### Also See

Microsoft Access ↑ Database-specific Notes

# Oracle

# Building a File with Create RRNO Column selected

For Oracle, in addition to having permission to alter the table, you require permission to create a sequence and a trigger as the table owner.

# **Removing X\_UPID and X\_RRNO Columns**

For Oracle, if you remove the columns, you should also remove the trigger and the sequence. Use the Oracle data dictionary views DBA\_TRIGGERS and DBA\_SEQUENCES (or USER\_TRIGGERS and USER\_SEQUENCES when you are logged on as the table owner) to find the trigger and sequence to be dropped. (The object names will contain the LANSA file name.)

## Also See

Create RRNO Column

↑ Database-specific Notes

## 4.10.3 IBM i RDMLX Other Files

This facility is NOT to be used to load a LANSA file. Instead, export/import the definition you want to use into the new partition. Do not compile the file. Set the user's library to access the File and IO Module.

The following information pertains to RDMLX Other Files. For RDML Other Files, refer to Load the Definition of a File NOT Maintained by LANSA from the *LANSA for i User Guide*.

When loading an IBM i Other File for RDMLX, LANSA supports a much wider range of field types. For details on the various field types, refer to Field Types in the *LANSA Technical Reference Guide*.

Database columns will be skipped during the load if they meet the following criteria:

- Type is Graphic, Vargraphic, DBCLOB with a non-Unicode CCSID. (Unicode CCSIDs are 1200 and 13488 for Graphic data.)
- Type is Rowid, Decfloat, XML, or CLOB with CCSID 1208.

DateTime fields may require special handling. (Refer to Handling of Timestamp or DateTime columns.)

As IBM i Other Files never have their external definition changed by LANSA, additional columns to support LOBs are never added. (Refer to No additional LOB columns.)

On Windows and Linux, IBM i Other Files are treated as if they are LANSA Files. Some IBM i Other Files that have features not supported by LANSA Files may not build or execute correctly on Windows and Linux.

#### Also See

Load Other File in the *Technical Reference Guide* How BLOB and CLOB columns are handled by RDMLX Other Files 1 4.10 Load Other File

# 4.10.4 Setting Other File Load Options

If appropriate, the option of the files being loaded can be changed via the Load options which are accessed when a file is selected in the Load Other File list.

🛤 🖪 🔎				
IBN i Other Data Sources				
Contect to database	Lange and the second se			
N	XYZ.			
Select to change the file options			(blank for all tables)	
i un or partial table fiame			(blank for an tables)	
			66 Find	
Table name 🔺 🕯	Description	Qualifier	Owner	
▷ □ x = DEPTAB		xyz1	EVDEXLIB	
▶ 🗖 🛶 🕞 DFRTAB		xyz2	XDFMOLIB	

Use this Load Options dialog to select the changed options.

8	Load Options	×
Submit compiles		
Allow read access only		
Add columns to support LOBs		
Create RRNO column		
Suppress IOM0034		
Commitment Control		
I/O Module Library	Default File Library (F)	*
	OK Cancel	

Details of these options are in Other Data Sources Load Options in the *Technical Reference Guide*.

# 4.11 LANSA Internal Tables

Loading and using LANSA internal tables is not supported by LANSA. The table layouts and the meaning of data values in these tables will change from time to time. LANSA does not publish the changes that have been made.

If you wish to provide features that LANSA does not currently provide by accessing the LANSA internal tables directly, you do so entirely at your own risk. When LANSA changes the LANSA internal tables it is entirely your responsibility to alter your application to behave appropriately.

Before you consider making changes to the LANSA internal tables, contact your product vendor for a copy of the table formats.

↑ 4.10 Load Other File

## **5. Creating Applications Using Functions**

Before creating your new LANSA application, review the following topics:

- 5.1 Getting Started with Function Development
- 5.2 Developing with Processes and Functions
- 5.3 Sample RDML Programs
- 5.4 Producing Reports Using LANSA

#### Also See

Processes and Functions in the *Technical Reference Guide* Editing Components and Function in the *User Guide* 

# **5.1 Getting Started with Function Development**

With LANSA's integrated development environment, building an application is a simple two step process:

- Step 1 Build the Repository
- Step 2 Build Your Application Programs.

Step 1 is described in Creating Fields and Creating Application Databases.

As you begin Step 2, you begin to create your application programs using LANSA processes and functions. Review the following topics:

5.1.1 Setting Standards & Practices

5.1.2 Process & Function Naming

- 5.1.3 RDML Practices
- 5.1.4 Process & Function Guidelines
- 5.1.5 Before You Begin Checklist

**Note:** As you begin programming, you may find additional logic that you wish to add to the repository. You often use a cyclic development process of refining repository contents as you start developing your programs. Many of the productivity and maintenance benefits are the result of how LANSA RDML uses the LANSA Repository. (Refer to Repository and RDML.)

It is also important to understand the characteristics of processes and how they interact with functions. (Refer to 5.2 Developing with Processes and Functions.)

### Also See

Processes and Functions in the LANSA Application Design Guide

↑ 5. Creating Applications Using Functions

# **5.1.1 Setting Standards & Practices**

It is very important to define and implement a set of development standards before you begin to build your application. Like the repository standards, defining and enforcing standards for RDML development will help to protect your application systems investment and will improve developer productivity.

Development standards should be defined for both process and function names. These standards will help to ensure that uniquely defined and identifiable program objects are created.

It is also important to establish a set of RDML coding practices. These practices should include a set of published guidelines which are reviewed and enforced. Setting practices such as code reviews are an excellent way to enforce standards and develop better RDML programmers. LANSA also provides a complexity monitoring facility which can monitor RDML programs when they are compiled.

When developing with LANSA for i and Visual LANSA in a distributed development environment, you will have a Master Repository on IBM i and local copies of the repository in Visual LANSA. In this case, it becomes even more important that proper procedures and standards are enforced. You should have a Change Management strategy defined that includes a plan for configuring task tracking.

The success of a project will be determined by the commitment of all staff to follow the standards created at your site.

For more details, refer to Processes and Functions in the LANSA Application Design Guide.

### Also See

5.2 Developing with Processes and Functions

↑ 5.1 Getting Started with Function Development

# **5.1.2 Process & Function Naming**

The following naming recommendations are provided for "in house" application development as opposed to vendor "package" development.

- Process should be in the format:**ooooPRnn**
- Functions should be in the format:**ooooaaa**
- The **oooo** part of the name indicates the basic or primary object that the process or function uses (based on the data model entity). For example, CUST Customer or ORDR- Order.
- The **nn** portion of the process name is a sequentially assigned number within the OOOO grouping. For example, CUSTPR01 Online Customer Maintenance, or CUSTPR02 Batch Customer Maintenance.
- The **aaa** portion of the function name indicates the action to be taken on the object by functions within the process. For example, MNT- maintenance, or REP- reports (use R01, R02, R03 for multiple reports).
- Names should not contain the '#' or '@' symbol.

We strongly recommend that you enforce a set of standards for object naming at your site. For more details, refer to Processes and Functions in the *LANSA Application Design Guide*.

Please review the LANSA Object Names in the LANSA Technical Reference *Guide*.

# **Platform Considerations**

- **IBM i:** A process name must be unique within the entire LANSA partition. A function name must be unique within the process it is created. It is possible to have two functions with the same name in a single partition if they are located in different processes.
- **Windows:** A process name must be unique within the entire LANSA system. All functions must be defined as type \*DIRECT. Functions names must be unique in the partition.

### $\uparrow$ 5.1 Getting Started with Function Development

# 5.1.3 RDML Practices

LANSA provides a powerful repository-based 4GL. The application architecture is based on a repository which shares and reuses information with the 4GL (RDML). Programming in a repository-based 4GL environment is not the same as coding with a 3GL. (Refer to LANSA RDML.)

It is possible to create 3GL style applications with any 4GL, but when you use 3GL techniques with a 4GL, you lose all of the benefits of the LANSA development environment. If you code validation rules in your programs or define all your variables within a function, this information cannot be shared or reused by other programs. Following are points to remember:

- Do not try to solve problems with 3GL approaches.
- Never translate 3GL code into 4GL code.
- Never translate 3GL "logic" into 4GL "logic".

It is important to focus on learning and implementing 4GL coding techniques and practices when building your LANSA applications.

Use the power of the combination of the repository and RDML. Remember that information in the repository can be shared and reused. The more information you place in the repository instead of the RDML programs, the more powerful your application becomes. Development productivity dramatically increases and maintenance efforts are significantly reduced.

Finally, two important, but often overlooked, aspects of designing, building and maintaining application systems are:

- The cost of developing a function rises very quickly as its complexity increases.
- The cost of maintaining a function rises even more quickly as its complexity increases.

Put simply, complexity costs! Do not over design or over build a function. Determine what functionality the user requires and the most efficient means of providing it. Expend your efforts where they will be most useful to the user. For more details, refer to the LANSA Application Design Guide.

↑ 5.1 Getting Started with Function Development

# **5.1.4 Process & Function Guidelines**

The following are some additional guidelines for working with processes and functions:

- Process and function names should follow site standards. Be sure to define, publish and enforce your site standards.
- LANSA RDML provides a set of FUNCTION characteristics which can be standardized and made part of application templates. Review and use the FUNCTION options. For example, all functions must be \*DIRECT if developing in a Visual LANSA environment.
- Use developer code reviews as a means of training new developers and as a means of checking that site standards are being followed.
- Use the LANSA application templates and customize these templates to meet your site standards. Templates are a quick way to ensure that all screens and reports have the same look and feel. (Refer to Using Application Templates.)
- Create help text for all processes and functions. Because of LANSA's development security, you can even have users enter help text.
- Use the impact analysis tool to help control changes and assess what changes need to be performed. Impact analysis can save a significant amount of time testing and rebuilding your application. Impact analysis can also create lists for group objects.
- No process should contain more than 20 functions, unless it is a special subroutine or prompt key "grouping" process.

For details about designing processes and functions, refer to Processes and Functions in the LANSA Application Design Guide.

### Also See

5.2 Developing with Processes and Functions

 $\uparrow$  5.1 Getting Started with Function Development

# **5.1.5 Before You Begin Checklist**

Before a single RDML process or function is coded, it is recommended that the following checklist has been satisfied:

- All repository development standards and guidelines are known by/available to every member of the project team.
- All LANSA Object naming standards and guidelines are known by/available to every member of the project team. (Refer to LANSA Object Names in the *LANSA Technical Reference Guide*.)
- All process and function development standards and guidelines are known by/available to every member of the project team. This should include task tracking procedures. (Refer to Using Task Tracking in LANSA in the *Administrator Guide*.)
- Principal repository development has been completed.
  - All known repository entries, validation rules and HELP text have been input.
  - All known system variables have been identified, implemented and tested. All members of the project team are aware of their existence and their use.
  - All known files in the database have been designed, implemented and are understood by all members of the project.
  - All file level validation checks have been implemented and tested.
- 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 functions. These have been tested and conform to all site standards for "look" and "feel". (Refer to Using Application Templates.)
- The entire application system has been broken down into identifiable processes and functions which will perform all of the required business functions (i.e. the whole system has been designed before coding commences).
- The complete backup and recovery strategy to be used by the application system has been designed.

• Finally, you should understand the concepts described in 5.2 Developing with Processes and Functions.

For details about designing processes and functions, refer to Processes and Functions in the LANSA Application Design Guide.

 $\uparrow$  5.1 Getting Started with Function Development

# **5.2 Developing with Processes and Functions**

As you begin to create new processes and functions for your application, you should be familiar with the following topics:

- 5.2.1 What is a Process?
- 5.2.2 Process Style
- 5.2.3 What is a Function?
- 5.2.4 How to Create Processes
- 5.2.5 How to Create Functions
- 5.2.6 Steps for Building Function Applications
- 5.2.7 Attached Processes/Functions Concepts
- 5.2.8 Function Control Table Concepts
- 5.2.9 Special Entries Concepts
- 5.2.10 Process Parameter Concepts
- 5.2.11 Action Bar Concepts

#### Also See

Processes and Functions in the *Technical Reference Guide* Editing Processes in the *User Guide* 

↑ 5. Creating Applications Using Functions

# 5.2.1 What is a Process?

In LANSA, a process is a group of related functions (programs). A process can be thought of as a container which holds RDML functions. It also provides an interface or menu to the functions it contains. Every LANSA function must be defined as part of a process. If you are developing with components (such as Forms, Reusable Parts or WAMs), you will not require a LANSA process.

When used in an interactive environment, a process appears as either a menu with a selection of user options, or as an action bar. (Refer to 5.2.2 Process Style.) A process can be thought of as an interpretive control table or menu used to access functions. It is not a "coded" program (it has no RDML), but it must be compiled. When used in a batch environment, a process usually consists of only one function or a group of closely related functions (such as related reports).

Following are some important characteristics of processes:

- Processes must have a name and a description. Process name should be a maximum of 8 characters.
- A process can be invoked as a menu.
- A process may call another process or a process may call a specific function within another process. (Refer to 5.2.7 Attached Processes/Functions Concepts.)
- A process may call other programs or issue commands. (Refer to 5.2.9 Special Entries Concepts.)
- A process is defined by its functions and control table. (Refer to 5.2.8 Function Control Table Concepts.)
- A process does not contain any coded RDML commands.
- A process must be compiled.

The design of the application system will determine how many processes are required and what functions will belong to a given process. The process/function structure in LANSA is an important part of modular design, which helps to build maintainable systems. (Refer to Processes and Functions in the *LANSA Application Design Guide.*)

# **Platform Considerations**

• **IBM i:** A process name must be unique within the entire LANSA partition. A function name must be unique within the process it is created. It is

possible to have two functions with the same name in a single partition if they are located in different processes.

• Windows: A process name must be unique within the entire LANSA system. All functions must be defined as type \*DIRECT. Functions names must be unique in the partition.

#### Also See

- 5.2.4 How to Create Processes
- $\uparrow$  5.2 Developing with Processes and Functions

# 5.2.2 Process Style

LANSA supports two process styles:

- SAA/CUA Compliant Menu
- Action Bar. Refer to 5.2.11 Action Bar Concepts for details.

Most of your applications will use the SAA/CUA Menu style. This menu style displays a menu item for each function in the process. The menu uses a simple interface to call programs.

The Action Bar style process displays the menu as an action bar with a series of categories and options. The CUA Basic Interface Design Guidelines define an action bar as follows:

- "The action bar is the panel element at the top of the panel that consists of a list of choices that represent a group of related actions that the user can request. A group of actions appears in a pull down menu when the users request an action bar choice. Pull downs are located immediately below the action bar.
- The actions typically affect the information displayed in the work area or in some way control the user's dialogue with the application."

 $\Uparrow$  5.2 Developing with Processes and Functions

# 5.2.3 What is a Function?

In LANSA, a function is a group of RDML commands. It is a set of RDML commands which are compiled into an executable program. For example, an interactive or online function might add or change data in the Employee file. The function uses a character-based presentation. A batch function might produce an employee report or update available employee records. Functions are procedural in nature.

Following are some important characteristics of functions:

- A function "belongs to" or "is contained in" a process. (Refer to 5.2.1 What is a Process?.)
- A function cannot be created without a process.
- Functions must have a name and description. Function names must be unique within processes and must be unique in a partition. Function names may be up to 7 characters.
- Functions can be created automatically using templates, or can be created manually, or can be copied from an existing function. (Copied functions must be in the same partition.)
- A function must be compiled before it can be used.
- Functions compiled using Visual LANSA must use the FUNCTION OPTION(\*DIRECT) command. Consequently, Visual LANSA function names must be unique in the partition.
- Functions support character-based presentation formats using RDML commands.
- A function must be Enabled for RDMLX to use RDMLX features. (For details, refer to RDML and RDMLX Partition Concepts in the *Administrator's Guide*.)

Just as in any computer program, modular or well structured code is important for maintenance. The developer is responsible for how functions are constructed. (Refer to 5.1.3 RDML Practices.)

# **Platform Considerations**

• **IBM i:** A process name must be unique within the entire LANSA partition. A function name must be unique within the process it is created. It is possible to have two functions with the same name in a single partition if they are located in different processes.

• Windows: A process name must be unique within the entire LANSA system. All functions must be defined as type \*DIRECT. Functions names must be unique in the partition.

#### Also See

- 5.2.5 How to Create Functions
- $\uparrow$  5.2 Developing with Processes and Functions

# 5.2.4 How to Create Processes

There are several ways of creating a process:

- 1. Manually define
- 2. Copy an existing process

**Option 1**. When manually defining a process, you must specify the name, description and type of process. Again the process name should follow your site standards and must be unique in the partition. The 5.2.2 Process Style can be one of the following:

- SAA/CUA (SAA/CUA compliant menu style)
- ACT/BAR (action bar style).

In most cases, you will use SAA/CUA processes.

**Option 2.** If you copy a process, the complete process definition will be copied. All of the functions "owned" by the process will be copied as well. Note: Function names must be unique in a partition. Copying a process will create a set of functions with identical names. These functions must be renamed or they will fail to compile.

#### Also See

Creating Processes in the User Guide Process Definition in the Technical Reference Guide

↑ 5.2 Developing with Processes and Functions

# 5.2.5 How to Create Functions

Once a process has been created, you may add functions to it. Functions can be created in several ways:

- 1. Generate from an application template
- 2. Copy an existing function
- 3. Manually define

Option 1. Templates can be used to generate complete functions or a part of a function. Almost all of your new functions should be created from a template to ensure site standard are followed. (Refer to Using Application Templates.)

Option 2. A function can be copied from any process in the partition. Both the process and function name can be prompted. Copying an existing function often provides an excellent starting point. If you regularly copy a standard function, you should consider creating a template for the function. When you copy an existing function, you must still compile the function before it can be executed.

Option 3. A manually defined function is created using the LANSA editor. Though you will use the editor to modify the RDML, in most cases you should begin with a template or copy an existing function. Even when you manually define your functions, you should still use building block templates.

### Also See

5.2.4 How to Create Processes
Creating Functions in the User Guide
Function Definitions in the Technical Reference Guide
↑ 5.2 Developing with Processes and Functions

# **5.2.6 Steps for Building Function Applications**

The basic development steps when building your LANSA applications using functions are as follows:

- 1. Create a process. (Refer to 5.2.4 How to Create Processes.)
- 2. Create the required function(s) within the process. (Refer to 5.2.5 How to Create Functions.)
- 3. Complete the RDML coding and testing of each function. (Refer to Process/Function Compile in the *Technical Reference Guide*.)
- Review the process characteristics to build online menus and to control function execution. (Refer to 5.2.7 Attached Processes/Functions Concepts, 5.2.8 Function Control Table Concepts, 5.2.9 Special Entries Concepts.)
- 5. Continue to create other processes and functions as required in your application design. (Refer to Processes and Functions in the *LANSA Application Design Guide*.)
- 6. Finalize the overall application interface by building your main application menu. (Refer to 5.2.7 Attached Processes/Functions Concepts, 5.2.8 Function Control Table Concepts, 5.2.9 Special Entries Concepts.)

If you are creating action bar style applications using LANSA functions, refer to 5.2.11 Action Bar Concepts.

For details about the deployment of LANSA applications, refer to the LANSA Application Deployment Tool Guide.

#### Also See

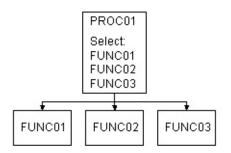
Processes and Functions in the *Technical Reference Guide* Editing Processes in the *User Guide* 

↑ 5.2 Developing with Processes and Functions

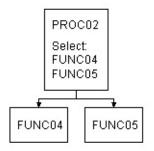
# **5.2.7 Attached Processes/Functions Concepts**

A process is essentially a menu of functions.

A process named PROC01 with 3 functions named FUNC01 -> FUNC03 could be visualized like this:

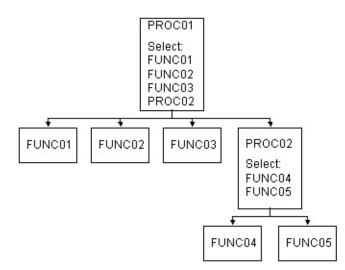


If another process named PROC02 existed and had 2 associated functions called FUNC04 and FUNC05, then it could be visualized like this:

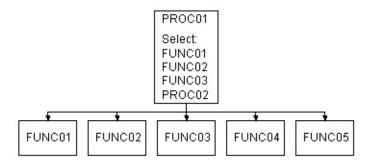


Both PROC01 and PROC02 can be independently invoked from within LANSA. However, it is possible to "attach" PROC02 to PROC01 by using the facilities discussed in the following sections.

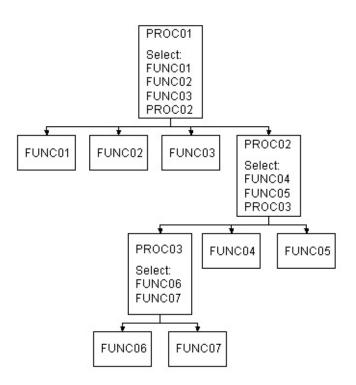
If PROC02 was "attached" to PROC01, then PROC01 would now be visualized like this:



It is also possible to "attach" FUNC04 and FUNC05 directly to PROC01 (even though they are part of process PROC02). If this was done PROC01 could be visualized like this:



In fact, process PROC02 may have a process(s) "attached" to it. For instance, if process PROC03 was attached to process PRC02 and process PROC02 was itself attached to process PROC01, then process PROC01 could be visualized like this:



It can be seen from these examples that by "attaching" a process to another process a "hierarchy" of process menus can be created.

Some other points to note about this facility are:

- Multiple processes may be attached to a single process.
- Multiple functions (from other processes) may be directly attached to a process.
- A process (or any of its associated functions) may be attached to itself.
- If a process B is attached to process A, then it is possible to attach process A to process B.
- Processes can be built into a "hierarchy" by using this facility. There is no limit to the "depth" of the hierarchy that can be defined, but when actually using a process the "depth" being used must not exceed 9 processes. If it does an error message will be issued indicating that it is not possible to go any deeper in the process "hierarchy" and that the required process should be accessed via a different route.

#### Warning

• When a process or function is deleted, all associated attachments are also automatically deleted.

# **Platform Considerations**

- **IBM i:** Processes and functions can be attached and detached dynamically. There is no need to recompile any associated functions after attaching or detaching them from another process.
- Windows: Process must be recompiled for changes to take effect.

#### Also See

Attached Process/Function Definitions in the *Technical Reference Guide* Attachments Tab in the *User Guide* 

 $\Uparrow$  5.2 Developing with Processes and Functions

# **5.2.8 Function Control Table Concepts**

The function control table is a table that controls how and when the various functions within a process are used and the ordering sequence that the functions should be in when displayed on the process menu.

Every function defined in a process has an entry in the function control table. It is automatically created with default values when the function is created and cannot be removed. However it can be changed from its default values to suit the specific requirements of a user application of LANSA.

Every entry in the function control table contains the following elements:

- The name of the function to which the entry belongs.
- The description of the function to which the entry belongs.
- An indication of whether or not the function should be displayed on (and therefore directly accessible from) the process's main menu.
- The Sequence that the Function should be ordered in on the process menu when displayed.
- The name of the associated "default next function".
- A list of up to 20 "allowable next functions".

The function control table associated with a process called EXAMPLE which has 3 functions called FUNC1, FUNC2 and FUNC3 would look like this before any user changes were made to it:

Function Name	Description	Display on Menu	Default Next Function	Allowable next functions
FUNC1	Function 1	YES	MENU	*ANY
FUNC2	Function 2	YES	MENU	*ANY
FUNC3	Function 3	YES	MENU	*ANY

In the section that dealt with "Creating a new function" a list of "reserved" function names was mentioned. This was a list of function names that were "reserved" for internal LANSA use and could not be used as valid user created function names:

Name	<b>Reserved meaning / description</b>	
MENU	Display process main menu	
EXIT	Exit from LANSA	
HELP	Display process HELP text	
SELECT	Select next function from list of allowable function	
EOJ	End all batch processing	
RETRN	Return control to calling process or function	
ERROR	Abort process with an error	
*ANY	Any function name	

Taking the function control table for process EXAMPLE and the list of reserved function names together it can be seen that what the table is actually specifying to LANSA is:

When using FUNC1 the most common (or default) next function is to display the process's main menu.

After using FUNC1 you can go to any other function in the process.

- Exactly the same conditions apply to using FUNC2 and FUNC3.
- All of the functions are to appear on the process's main menu and are therefore directly and independently accessible from the menu.

However, if the function control table for process EXAMPLE was changed as follows:

Function Name	Description	Display on Menu	Default Next Function	Allowable next functions
FUNC1	Function 1	YES	FUNC2	FUNC2
FUNC2	Function 2	NO	FUNC3	FUNC3 FUNC1 SELECT
FUNC3	Function 3	NO	MENU	MENU

The following conditions now apply:

- Once FUNC1 is used, it is only possible to go to FUNC2.
- In FUNC2, we can only go to FUNC3 or back to FUNC1 and the usual thing to do next is to go to FUNC3. In addition, the "reserved" function SELECT can be used. This will result in a list of the allowable next functions (FUNC1 and FUNC3) being displayed as a "sub menu". From this you can select the desired next function.
- In FUNC3, we can only go back and re-display the process's main menu.
- Only FUNC1 is to appear on the process's main menu. Thus only FUNC1 can be accessed from the process main menu. FUNC2 can only be accessed from FUNC1 and FUNC3 can only be accessed if FUNC2 (and therefore FUNC1) have been accessed first.

Hopefully this example illustrates the main purpose of the function control table which is to provide a "path" through the functions.

The path may be "enforced" (e.g.: From FUNC1 you must go to FUNC2) or it may be "suggested" (e.g.: from FUNC2 you would normally go to FUNC3, but can go to FUNC1 if desired).

If the partition is a multilingual partition, you can specify language-specific descriptions by selecting the appropriate language in the Language list in this dialog box.

### Also See

Function Control Table Definitions in the *Technical Reference Guide* 

Functions Tab in the User Guide

↑ 5.2 Developing with Processes and Functions

# **5.2.9 Special Entries Concepts**

Special entries on the process menu are special operating system commands that have been defined to LANSA using this facility. For IBM i, these may be CL (control language) commands. For Windows, these may be simple DOS commands or program execution commands.

For Windows, the special entry will be submitted to a CMD.EXE window for execution. You are responsible to ensure that a proper path and parameters have been specified to enable the command to execute. For example, the notepad editor could be opened by using the NOTEPAD.EXE command. A directory of the C drive could be written to a text file using a "DIR C:\ > C:\textfile.txt" command.

For IBM i, CL command that can be executed via IBM supplied programs QCMDEXC or QCAEXEC can be included onto a processes main menu. The command can be prompted at the time it is being used to allow you to specify or change the associated parameters. CL commands that allow users to display or control their own jobs are most commonly included onto the processes main menu. Some of these include:

WRKSBMJOB	Work with submitted jobs
WRKJOB	Work with current job
WRKOUTQ	Work with output queue
WRKJOBQ	Work with job queue
DSPMSG	Display message queue
SNDMSG	Send a message

For more information about the CL commands available on the IBM i and the programs QCMDEXC and QCAEXEC that are used to execute them, refer to the appropriate IBM supplied manual.

# **Platform Considerations**

The use of special entries may not be portable between operating systems. For example, IBM i CL commands cannot be executed on a Windows platform. Use caution with special entries if you are building applications for more than one

platform.

### Also See

Special Entries Definitions in the *Technical Reference Guide* Special Entries Tab in the *User Guide* 

 $\uparrow$  5.2 Developing with Processes and Functions

# **5.2.10 Process Parameter Concepts**

Process Parameters should not be used as they cannot be executed in Visual LANSA.

 $\Uparrow$  5.2 Developing with Processes and Functions

# 5.2.11 Action Bar Concepts

The action (menu) bar is a list of choices in the top part of a window used to navigate between different functions. When you click on a menu title, the menu drops down to show the menu options.

The action bar style was implemented to support a menu bar approach for character based terminals (non-graphical devices). Note: LANSA supports fully graphical menu bar application by developing application using graphical forms and components.

The action bar implementation within LANSA is at the process level. A process used must be in an SAA/CUA partition.

You can define a process as an action bar by editing the process definition.

A single line menu bar process might appear like this:

```
+-----+
|File Edit Options Help |
+-----+
```

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

```
+-----+

| File Edit Options Help |

+-----+

| 1. Open |

| 2. Close |

| 3. DTX to H/O |

| 4. FAX to H/O |

+----+
```

A menu bar can use from 1 to 3 lines across the top of the panel. Each line can contain at most 6 menu options, thus totaling 18 choices within one process.

Each menu option can have an associated pull down containing up to 9 options. Thus  $18 \ge 9 = 162$  choices can be controlled from one menu bar.

When a process is defined as menu bar, details about the layout and options in the menu bar and it's associated pull downs must be specified in the "Action Bar Control Table".

# **Platform Considerations**

**IBM i:** If you are defining an action bar for an IBM i application, it is

recommended that you review the information in The Action Bar Control Table the *LANSA for i User Guide*. The *LANSA for i User Guide* is written specifically for IBM i applications.

## Also See

Action Bar Table Definitions in the *Technical Reference Guide* SET\_ACTION\_BAR Built-In Function in the *Technical Reference Guide* Action Bar Table Tab in the *User Guide* 1 5.2 Developing with Processes and Functions

## **5.3 Sample RDML Programs**

The following RDML function samples are provided:

- 5.3.1 Adding Records to a File (One at a Time)
- 5.3.2 A Simple Inquiry
- 5.3.3 Modifying the Simple Inquiry to Do Maintenance
- 5.3.4 Modifying the Simple Inquiry to Do It All
- 5.3.5 Browsing and Selecting
- 5.3.6 A Multiple File Inquiry
- 5.3.7 Adding Records to a File (Many At a Time)
- 5.3.8 Minimizing Database I/Os
- 5.3.9 Using the KEEP\_LAST Parameter
- 5.3.11 Page At a Time Scrolling (Forward Only)
- 5.3.12 Page at a Time Scrolling (Backwards & Forwards)

# 5.3.1 Adding Records to a File (One at a Time)

Construct a function to add records to one file.

## **Files Involved**

Physical file CUSMST (customer master file)

#### **RDML Program**

GROUP\_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDL1 #ADDL2 #ADDL3)

```
BEGIN_LOOP
CHANGE FIELD(#CUSTOMER) TO(*DEFAULT)
REQUEST FIELDS(#CUSTOMER) EXIT_KEY(*YES *EXIT)
MENU_KEY(*YES *MENU)
INSERT FIELDS(#CUSTOMER) TO_FILE(CUSMST)
VAL_ERROR(*LASTDIS)
END_LOOP
```

#### **Points to Note:**

- Program keeps adding records until the EXIT or MENU function key is used (BEGIN\_LOOP and END\_LOOP commands).
- The EXIT and MENU function keys are enabled on the REQUEST screen. If either key is used the function will end. The values shown are defaults and would not normally be shown.
- The VAL\_ERROR parameter of the INSERT command is a default and would not normally be shown. If a validation error (file or dictionary level) is detected by the INSERT command the last display (the REQUEST command) will receive control to display the error details for correction.

1 5.3 Sample RDML Programs

# 5.3.2 A Simple Inquiry

Construct a function to display records from a file.

## **Files Involved**

Physical file CUSMST (customer master file)

### **RDML Program - Version 1**

GROUP\_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADD #ADDL2 #ADDL3) BEGIN\_LOOP L10 REQUEST FIELD(#CUSTNO) FETCH FIELDS(#CUSTOMER) FROM\_FILE(CUSMST) WITH\_KEY(#CUSTNO) NOT\_FOUND(L10) ISSUE\_MSG(\*YES) DISPLAY FIELDS(#CUSTOMER) END\_LOOP

## **RDML Program - Version 2**

GROUP\_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADD #ADDL2 #ADDL3) BEGIN\_LOOP REQUEST FIELD(#CUSTNO) FETCH FIELDS(#CUSTOMER) FROM\_FILE(CUSMST) WITH\_KEY(#CUSTNO) IF\_STATUS IS(\*OKAY) DISPLAY FIELDS(#CUSTOMER) ELSE MESSAGE MSGTXT('No customer exists with this number') ENDIF END\_LOOP

#### **Points to Note:**

- Both versions keep doing inquiries until the EXIT or MENU function key is used (BEGIN\_LOOP and END\_LOOP commands).
- The EXIT and MENU function keys are enabled on the REQUEST and DISPLAY screens by default. If either key is used the function will end.
- The first version uses the NOT\_FOUND and ISSUE\_MSG parameters of the

FETCH command to automatically cause a "not found" message to be issued and then to return control to the request command.

- The second version uses the IF\_STATUS command to check if the record was found. If it was it is displayed, else a message is issued that will appear on line 22/24 of the REQUEST screen
- 1 5.3 Sample RDML Programs.

# 5.3.3 Modifying the Simple Inquiry to Do Maintenance

Modify the previous inquiry function to also maintain records in the file.

## **Files Involved**

Physical file CUSMST (customer master file)

### **RDML Program**

GROUP\_BY NAME(#CUSTOMER) FIELDS((#CUSTNO \*NOCHG) #NAME #ADDL1 #ADDL2 #ADDL3)

BEGIN\_LOOP

```
REQUEST FIELD(#CUSTNO)
FETCH FIELDS(#CUSTOMER) FROM_FILE(CUSMST)
WITH_KEY(#CUSTNO)
```

```
IF_STATUS IS(*OKAY)
SET_MODE TO(*DISPLAY)
DISPLAY FIELDS(#CUSTOMER) CHANGE_KEY(*YES)
```

```
IF_MODE IS(*CHANGE)
UPDATE FIELDS(#CUSTOMER) IN_FILE(CUSMST)
VAL_ERROR(*LASTDIS)
ENDIF
```

ELSE

```
MESSAGE MSGTXT('No customer exists with this number') ENDIF
```

END\_LOOP

#### **Points to Note:**

- The function now supports customer inquiries and maintenance.
- Program keeps doing inquire/maintain until the EXIT or MENU function key is used (BEGIN\_LOOP and END\_LOOP commands).
- The EXIT and MENU function keys are enabled on the REQUEST, DISPLAY and POP\_UP screens by default. If either key is used the function

will end.

- In addition the CHANGE function key is enabled on the DISPLAY screen. If the CHANGE function key is used, the mode will be altered to \*CHANGE, and the screen re-presented with all fields (except #CUSTNO) input capable. The reason that #CUSTNO does not become input capable is that it has the \*NOCHG attribute in the GROUP\_BY command.
- After the DISPLAY has finished the mode is tested. If it is \*CHANGE then the user has used the CHANGE function key and altered the customer details. In this case the UPDATE is executed to update the customer master file. If the UPDATE command detects a validation error (file or dictionary level) the DISPLAY screen will be re-displayed with error details. This is because of the VAL\_ERROR(\*LASTDIS) parameter.
- Note that the UPDATE command does not specify a WITH\_KEY parameter to nominate the file record that is to be updated. This tells LANSA that the last record that was read from the file should be updated. Thus the record read by the FETCH command will be updated.
- LANSA does **NOT** lock the file record between the FETCH and the UPDATE command (although it could if the LOCK(\*YES) parameter was used).

So the possibility arises that another user could update the record in the interval between the FETCH and the UPDATE. This interval could be 30 minutes if the user decided to go to lunch.

If file CUSMST is created and maintained by LANSA this will not be a problem because the UPDATE command automatically checks for a change to the record in the interval between reading it and updating it. If another user has changed the record in the interval ..... the UPDATE command will act exactly as if a validation error had occurred and automatically issue a message indicating that the update was rejected. This will cause the DISPLAY screen to be re-displayed with an error message.

↑ 5.3 Sample RDML Programs

# 5.3.4 Modifying the Simple Inquiry to Do It All

Modify the previous inquiry function to inquiry, addition and deletion of records in the file.

# **Files Involved**

Physical file CUSMST (customer master file)

## **RDML Program**

GROUP\_BY NAME(#CUSTADD) FIELDS(#CUSTNO #NAME #ADDL1

END\_LOOP

#### **Points to Note:**

- The function supports display, add, maintain and delete of records from the file.
- Another strategy would be to use two separate functions and use the TRANSFER command to transfer control between them.
- 1 5.3 Sample RDML Programs

#### Function ADD - To Add New Records to CUSMST

GROUP\_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDL1 #ADDL2 #ADDL3) **BEGIN\_LOOP** REQUEST FIELDS(#CUSTOMER) EXIT KEY(\*YES \*EXIT) MENU KEY(\*YES \*MENU) INSERT FIELDS(#CUSTOMER) TO FILE(CUSMST) VAL ERROR(\*LASTDIS) END LOOP Function MAINT - To Display / Maintain CUSMST Records GROUP BY NAME(#CUSTOMER) FIELDS((#CUSTNO \*NOCHG) #NAM #ADDL1 #ADDL2 #ADDL3) **BEGIN LOOP** SET MODE TO(\*DISPLAY) REQUEST FIELD(#CUSTNO) ADD KEY(\*YES) IF MODE IS(\*ADD) TRANSFER TOFUNCTION(ADD) EXCHANGE(#CUSTNO) ENDIF FIELDS(#CUSTOMER) FROM FILE(CUSMST) FETCH WITH KEY(#CUSTNO) IF STATUS IS(\*OKAY) DISPLAY FIELDS(#CUSTOMER) CHANGE KEY(\*YES) DELETE KEY(\*YES) IF MODE IS(\*CHANGE) UPDATE FIELDS(#CUSTOMER) IN FILE(CUSMST) VAL ERROR(\*LASTDIS) ENDIF IF MODE IS(\*DELETE) DELETE FROM FILE(CUSMST) VAL ERROR(\*LASTDIS) **ENDIF** ELSE MESSAGE MSGTXT('No customer exists with this number') **ENDIF** END LOOP ↑ 5.3.4 Modifying the Simple Inquiry to Do It All

# **5.3.5 Browsing and Selecting**

Create a function to display all customers whose names generically match the name specified. Allow customers to be chosen from the list for a detailed display.

## **Files Involved**

Physical file CUSMST (customer master file) and logical file CUSMSTV1 which is a view of CUSMST viewed by customer name.

## Version 1

```
GROUP_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDL1
#ADDL2 #ADDL3)
```

DEF\_LIST NAME(#BROWCUST) FIELDS((#CHOOSE \*SELECT) #CUSTNO #NAME) COUNTER(#NUMCUSTS)

BEGIN\_LOOP

<< Get the customer name >>

REQUEST FIELD(#NAME)

<< Build up list of customers with same generic name >>

```
CLR_LIST NAMED(#BROWCUST)
SELECT FIELDS(#BROWCUST) FROM_FILE(CUSMSTV1)
WITH_KEY(#NAME) GENERIC(*YES)
ADD_ENTRY TO_LIST(#BROWCUST)
ENDSELECT
```

<< If none found issue message >>

```
IF COND('#NUMCUSTS = 0')
MESSAGE MSGTXT('No customers found with this name')
ELSE
```

<< else display list for selection >>

DISPLAY BROWSELIST(#BROWCUST)

<< process selected customers and display in detail >>

```
SELECTLIST NAMED(#BROWCUST) GET_ENTRYS(*SELECT)
FETCH FIELDS(#CUSTOMER) FROM_FILE(CUSMST)
WITH_KEY(#CUSTNO)
DISPLAY FIELDS(#CUSTOMER)
ENDSELECT
```

ENDIF

END\_LOOP

#### **Points to Note:**

• The GENERIC(\*YES) option automatically causes generic searching to be performed against the customer name.

## Version 2

GROUP\_BY NAME(#CUSTOMER) FIELDS(#CUSTNO #NAME #ADDL1 #ADDL2 #ADDL3)

DEF\_LIST NAME(#BROWCUST) FIELDS((#CHOOSE \*SELECT) #CUSTNO #NAME) COUNTER(#NUMCUSTS)

DEFINE FIELD(#GENNAME) REFFLD(#NAME)

BEGIN\_LOOP

<< Display the list - empty on first cycle >>

DISPLAY FIELD((#GENNAME \*INPUT)) BROWSELIST(#BROWCUST)

<< Process any selected entries - none in first cycle >>

CHANGE FIELD(#TOTSELECT) TO(0)

SELECTLIST NAMED(#BROWCUST) GET\_ENTRYS(\*SELECT) CHANGE FIELD(#TOTSELECT) TO('#TOTSELECT + 1') FETCH FIELDS(#CUSTOMER) FROM\_FILE(CUSTMST) WITH\_KEY(#CUSTNO) DISPLAY FIELDS(#CUSTOMER) ENDSELECT

<< If none selected build a new list >>

IF COND('#TOTSELECT = 0')

CLR\_LIST NAMED(#BROWCUST) SELECT FIELDS(#BROWCUST) FROM\_FILE(CUSMSTV1) WITH\_KEY(#GENNAME) GENERIC(\*YES) ADD\_ENTRY TO\_LIST(#BROWCUST) ENDSELECT

IF COND('#NUMCUSTS = 0') MESSAGE MSGTXT('No customers found with this name') ENDIF

ENDIF

END\_LOOP

#### **Points to Note:**

- When the BEGIN\_LOOP / END\_LOOP loop is entered for the first time the list will be empty so it will not display at the first DISPLAY command. Likewise the SELECTLIST loop will not process any entries .... so a search will be made for the customers.
- Subsequent iterations of the loop will only re-build the customer list if no customer is chosen from the list for detailed display.
- The main difference between this version and the first is that this function only uses one screen format to ask for the customer name and to display the list. The first version uses 2 separate screen formats.
- Note the attribute \*INPUT associated with field #GENNAME. This states that field #GENNAME should always be input capable, no matter what screen processing mode is in use.

1 5.3 Sample RDML Programs

## 5.3.6 A Multiple File Inquiry

Create a function to display full details of an order.

### **Files Involved**

Physical files ORDHDR (order header), ORDLIN (order lines), CUSMST (customer master) and PROMST (product master).

### **RDML Program**

GROUP\_BY NAME(#ORDERHEAD) FIELDS(#ORDER #DATDUE #CL #NAME #ADDL1)

DEF\_LIST NAME(#ORDERLINE) FIELDS(#ORDLIN #PRODNO #DES #QTY #VALUE)

BEGIN\_LOOP

<< Get order number and fetch header and customer details >>

L10 REQUEST FIELDS(#ORDER) FETCH FIELDS(#ORDERHEAD) FROM\_FILE(ORDHDR) WITH\_KEY(#ORDER) NOT\_FOUND(L10) ISSUE\_MSG(\*YES) FETCH FIELDS(#ORDERHEAD) FROM\_FILE(CUSTMST) WITH\_KEY(#CUSTNO) NOT\_FOUND(L10) ISSUE\_MSG(\*YES)

<< Build list of associated lines and product details >>

CLR\_LIST NAMED(#ORDERLINE) SELECT FIELDS(#ORDERLINE) FROM\_FILE(ORDLIN) WITH\_KEY(#ORDER) FETCH FIELDS(#ORDERLINE) FROM\_FILE(PROMST) WITH\_KEY(#PRODNO) ADD\_ENTRY TO\_LIST(#ORDERLINE) ENDSELECT

<< Display full details of the order >>

DISPLAY FIELDS(#ORDERHEAD) BROWSELIST(#ORDERLINE)

END\_LOOP

# 1 5.3 Sample RDML Programs

## 5.3.7 Adding Records to a File (Many At a Time)

#### **Files Involved**

Physical file TARGET (sales targets file) that has the following fields:

Field	Туре	Length	Description
YEAR	S	2,0	Year
MONTH	S	2,0	Month
BRAND	А	5	Brand number / identity
QUANTITY	Р	9,0	Target sales quantity
DOLLARS	Р	11,2	Target sales dollar value

#### **RDML Program**

<< Define work fields for this function >>

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

<< Define group to appear on top of screen >>

GROUP\_BY NAME(#TIME) FIELDS(#YEAR #MONTH)

<< Define group that contains all fields to be inserted >>

GROUP\_BY NAME(#TARGET) FIELDS(#YEAR #MONTH #BRAND #QUANTITY #DOLLARS)

<< Define list to be used for data entry >>

DEF\_LIST NAME(#VALUES) FIELDS(#BRAND #QUANTITY #DOLLAF

<< Loop until EXIT or MENU function key used >>

#### BEGIN\_LOOP

<< Clear screen and initialize list with 20 entries >>

CHANGE FIELD(#TARGET) TO(\*NULL) SET\_MODE TO(\*ADD) INZ\_LIST NAMED(#VALUES) NUM\_ENTRYS(20)

<< Accept input until no errors exist in data >>

DOUNTIL COND('#ERRCOUNT = 0')

SET\_MODE TO(\*ADD) DISPLAY FIELDS(#TIME) BROWSELIST(#VALUES)

<< Process all not "null" and not "accepted" records >>

FIELD(#ERRCOUNT) TO(0) CHANGE SELECTLIST NAMED(#VALUES) GET\_ENTRYS(\*NOTNULL \*ADD) INSERT FIELDS(#TARGET) TO\_FILE(TARGET) VAL\_ERROR(L10) L10 IF STATUS IS\_NOT(\*OKAY) CHANGE FIELD(#ERRCOUNT) TO('#ERRCOUNT + 1') SET\_MODE TO(\*ADD) ELSE SET\_MODE TO(\*DISPLAY) ENDIF UPD\_ENTRY IN\_LIST(#VALUES) **ENDIF ENDSELECT** 

ENDUNTIL

END\_LOOP

#### **Points to Note:**

• Program keeps adding records until the EXIT or MENU function key is used

(BEGIN\_LOOP and END\_LOOP commands).

- The EXIT and MENU function keys are enabled on the DISPLAY screen. If either key is used the function will end.
- Although the fields #YEAR and #MONTH are in every file record the user only has to specify them once .... at the top of the screen. Thus the user is entering "batches" of values for a particular year and month. The actual "values" are specified in the list displayed at the bottom of the screen. Up to 20 may be entered at one time.
- Note how the screen "mode" is set before the list is initialized with 20 "null" records. Since the mode is set to \*ADD these can be used for data entry.
- The SELECTLIST / ENDSELECT processing loop only selects list entries that are not "null" (parameter \*NOTNULL) and that are also in "add" mode (parameter \*ADD). Thus only entries from the list that the user has actually entered data into, and have not already been inserted into the file, are processed.
- The VAL\_ERROR parameter of the INSERT command causes control to be passed to the next command if a validation error is detected. The next command then examines the I/O "status". If any error occurred the error count is incremented and the mode set to "add". If no error occurred the mode is set to "display".
- After the mode has been set to \*ADD or \*DISPLAY (by the commands associated with the preceding point) the list entry is updated. If the mode is \*DISPLAY when the update occurs the list entry will no longer be input capable and can no longer be changed by the user (because it has been inserted into the file). If the mode is \*ADD then the fields in error will be reverse imaged by the update command and the user will be requested to correct the error.

The fact that a record has been "accepted" or "rejected" by the INSERT command can be indicated to the user by making the following modifications to the RDML commands:

<< Define work fields for this function >>

DEFINE FIELD(#ERRCOUNT) TYPE(\*DEC) LENGTH(7) DECIMALS(( DEFINE FIELD(#ACCEPTED) TYPE(\*CHAR) LENGTH(3) COLHDG('Target' 'has been' 'Accepted')

<< Define group to appear on top of screen >>

#### GROUP\_BY NAME(#TIME) FIELDS(#YEAR #MONTH)

<< Define group that contains all fields to be inserted >>

GROUP\_BY NAME(#TARGET) FIELDS(#YEAR #MONTH #BRAND #QUANTITY #DOLLARS)

<< Define list to be used for data entry >>

DEF\_LIST NAME(#VALUES) FIELDS(#BRAND #QUANTITY #DOLLARS (#ACCEPTED \*OUTPUT))

<< Loop until EXIT or MENU function key used >>

BEGIN\_LOOP

<< Clear screen and initialize list with 20 entries >>

CHANGE FIELD(#TIME) TO(\*NULL) CHANGE FIELD(#VALUES) TO(\*NULL) SET\_MODE TO(\*ADD) INZ\_LIST NAMED(#VALUES) NUM\_ENTRYS(20)

<< Accept input until no errors exist in data >>

DOUNTIL COND('#ERRCOUNT = 0')

SET\_MODE TO(\*ADD) DISPLAY FIELDS(#TIME) BROWSELIST(#VALUES)

<< Process all not "null" and not "accepted" records >>

CHANGE FIELD(#ERRCOUNT) TO(0) SELECTLIST NAMED(#VALUES) GET\_ENTRYS(\*NOTNULL \*ADD) INSERT FIELDS(#TARGET) TO\_FILE(TARGET) VAL\_ERROR(L10) L10 IF\_STATUS IS\_NOT(\*OKAY) CHANGE FIELD(#ERRCOUNT) TO('#ERRCOUNT + 1') SET\_MODE TO(\*ADD) ELSE SET\_MODE TO(\*DISPLAY) CHANGE FIELD(#ACCEPTED) TO(YES) ENDIF UPD\_ENTRY IN\_LIST(#VALUES) ENDIF ENDSELECT

ENDUNTIL

END\_LOOP

The main processing loop of the program could be modified to allow updates to "accepted" records like this: << Define work fields for this function >>

DEFINE FIELD(#ERRCOUNT) TYPE(\*DEC) LENGTH(7) DECIMALS(( DEFINE FIELD(#ADD\_RRN) TYPE(\*DEC) LENGTH(15) DECIMALS(( DEFINE FIELD(#ACCEPTED) TYPE(\*CHAR) LENGTH(3) COLHDG('Target' 'has been' 'Accepted')

<< Define group to appear on top of screen >>

GROUP\_BY NAME(#TIME) FIELDS(#YEAR #MONTH)

<< Define group that contains all fields to be inserted >>

GROUP\_BY NAME(#TARGET) FIELDS(#YEAR #MONTH #BRAND #QUANTITY #DOLLARS)

<< Define list to be used for data entry >>

DEF\_LIST NAME(#VALUES) FIELDS(#BRAND #QUANTITY #DOLLAF (#ACCEPTED \*OUTPUT)(#ADD\_RRN \*HIDDEN))

<< Loop until EXIT or MENU function key used >>

BEGIN\_LOOP

<< Clear screen and initialize list with 20 entries >>

CHANGE FIELD(#TIME) TO(\*NULL) CHANGE FIELD(#VALUES) TO(\*NULL) SET\_MODE TO(\*ADD) INZ\_LIST NAMED(#VALUES) NUM\_ENTRYS(20)

DOUNTIL COND('#ERRCOUNT = 0')

SET\_MODE TO(\*ADD) DISPLAY FIELDS(#TIME) BROWSELIST(#VALUES)

<< Process all not "null" records >>

CHANGE FIELD(#ERRCOUNT) TO(0) SELECTLIST NAMED(#VALUES) GET\_ENTRYS(\*NOTNULL)

IF COND('#ACCEPTED = YES')
UPDATE FIELDS(#TARGET) IN\_FILE(TARGET) VAL\_ERROR(L10)
WITH\_RRN(#ADD\_RRN)
ELSE
INSERT FIELDS(#TARGET) TO\_FILE(TARGET)

VAL\_ERROR(L10) RETURN\_RRN(#ADD\_RRN) ENDIF

L10 IF\_STATUS IS\_NOT(\*OKAY) CHANGE FIELD(#ERRCOUNT) TO('#ERRCOUNT + 1') ELSE CHANGE FIELD(#ACCEPTED) TO(YES) ENDIF

```
SET_MODE TO(*ADD)
UPD_ENTRY IN_LIST(#VALUES)
ENDIF
ENDSELECT
```

ENDUNTIL

END\_LOOP

↑ 5.3 Sample RDML Programs

## 5.3.8 Minimizing Database I/Os

Initial Example First Improved Example Second Improved Example

### **Files Involved**

Physical file TRANS (transactions file) has the following fields:

Field	Туре	Length	Description
TRANNUM	S	5	Transaction number
TRANTIME	S	6	Transaction time
TRANDATE	S	6	Transaction date
TRANTYPE	А	2	Transaction type
TRANUSER	А	10	Transaction user
TRANSTATE	А	3	Transaction state

and a file called TRNTYP (transaction types/descriptions) has the following fields:

Field	Туре	Length	Description
TRANTYPE	А	2	Transaction type
TRANDESC	А	20	Transaction full description

and a file called STATES (state codes/descriptions) has the following fields:

Field	Туре	Length	Description
STATCODE	А	3	State code
STATDESC	А	20	State full description

↑ 5.3 Sample RDML Programs

## **Initial Example**

A simple RDML program to list off the transactions file and also show full transaction and state descriptions fields might be coded like this:

```
GROUP_BY NAME(#TRANS) FIELDS(#TRANNUM #TRANDATE #TRA
#TRANTYPE #TRANDESC #TRANUSER #TRANSTATE #STATDES
```

SELECT FIELDS(#TRANS) FROM\_FILE(TRANS) FETCH FIELDS(#TRANS) FROM\_FILE(STATES) WITH\_KEY(#TRANSTATE) FETCH FIELDS(#TRANS) FROM\_FILE(TRNTYP) WITH\_KEY(#TRANTYPE) UPRINT FIELDS(#TRANS) ENDSELECT

If file TRANS contained 10,000 records this RDML program would do (10,000 + 10,000 + 10,000) = 30,000 database accesses.

However, by using a working list, or the KEEP\_LAST parameter on the FETCH commands, the speed of this program can be improved considerably.

↑ 5.3.8 Minimizing Database I/Os

## **First Improved Example**

The first method of improving the performance of this RDML program by using working lists involved loading the entire STATES file into a list when the program first starts and then replacing the FETCH command with a LOC\_ENTRY command:

GROUP\_BY NAME(#TRANS) FIELDS(#TRANNUM #TRANDATE #TRA #TRANTYPE #TRANDESC #TRANUSER #TRANSTATE #STATDES

DEF\_LIST NAME(#STATES) FIELDS(#STATCODE #STATDESC) TYPE(\*WORKING) ENTRYS(10)

SELECT FIELDS(#STATES) FROM\_FILE(STATES) ADD\_ENTRY TO\_LIST(#STATES) ENDSELECT

SELECT FIELDS(#TRANS) FROM\_FILE(TRANS) LOC\_ENTRY IN\_LIST(#STATES) WHERE('#STATCODE = #TRANSTATE FETCH FIELDS(#TRANS) FROM\_FILE(TRNTYP) WITH\_KEY(#TRANTYPE) UPRINT FIELDS(#TRANS) ENDSELECT

In the case where file TRANS contained 10,000 records and file STATES contained 6 records, this program would now do (10,000 + 6 + 10,000) = 20,006 database accesses, a net improvement of 9,994 less I/Os. The run time would probably be about 2/3 of the original version of the program.

↑ 5.3.8 Minimizing Database I/Os

### Second Improved Example

A further improvement can be made by using "demand" subroutines that exactly emulate the FETCH commands. This could be done like this:

GROUP\_BY NAME(#TRANS) FIELDS(#TRANNUM #TRANDATE #TRA #TRANTYPE #TRANDESC #TRANUSER #TRANSTATE #STATDES

```
SELECT FIELDS(#TRANS) FROM_FILE(TRANS)
EXECUTE SUBROUTINE(GET_STATE) WITH_PARMS(#TRANSTATE)
EXECUTE SUBROUTINE(GET_TYPE) WITH_PARMS(#TRANTYPE)
UPRINT FIELDS(#TRANS)
ENDSELECT
```

```
SUBROUTINE NAME(GET_STATE) PARMS((#GETSTATE *RECEIVED))
DEFINE FIELD(#GETSTATE) REFFLD(#STATCODE)
DEF_LIST NAME(#STATES) FIELDS(#STATCODE #STATDESC)
TYPE(*WORKING) ENTRYS(10)
LOC_ENTRY IN_LIST(#STATES) WHERE('#STATCODE = #GETSTATE')
IF_STATUS IS_NOT(*OKAY)
FETCH FIELDS(#STATES) FROM_FILE(STATES)
WITH_KEY(#GETSTATE)
ADD_ENTRY TO_LIST(#STATES)
ENDIF
ENDROUTINE
SUBROUTINE NAME(GET_TYPE) PARMS((#GETTYPE *RECEIVED))
```

```
DEFINE FIELD(#GETTYPE) REFFLD(#TRANTYPE)
DEF_LIST NAME(#TRNTYP) FIELDS(#TRANTYPE #TRANDESC)
TYPE(*WORKING) ENTRYS(50)
LOC_ENTRY IN_LIST(#TRNTYP) WHERE('#TRANTYPE = #GETTYPE')
IF_STATUS IS_NOT(*OKAY)
FETCH FIELDS(#TRNTYP) FROM_FILE(TRNTYP) WITH_KEY(#GET
ADD_ENTRY TO_LIST(#TRNTYP)
ENDIF
ENDROUTINE
```

Since these subroutines are "demand" driven the number of I/Os done depends upon the number of different state and type codes found in the TRANS files. If the TRANS file contained 3 different state codes and 5 different transaction types then in the case where 10,000 TRANS records were processed the RDML program would do (10,000 + 3 + 5) = 10,008 database accesses. This is a net saving of 19,992 I/Os. This version of the program would probably run in about 1/3 of the time of the original version.

↑ 5.3.8 Minimizing Database I/Os

### 5.3.9 Using the KEEP\_LAST Parameter

The initial version of this program can be made just as efficient as the second improved example by including the KEEP\_LAST parameter on the FETCH commands like this:

GROUP\_BY NAME(#TRANS) FIELDS(#TRANNUM #TRANDATE #TRA #TRANTYPE #TRANDESC #TRANUSER #TRANSTATE #STATDES

SELECT FIELDS(#TRANS) FROM\_FILE(TRANS) FETCH FIELDS(#TRANS) FROM\_FILE(STATES) WITH\_KEY(#TRANSTATE) KEEP\_LAST(6) FETCH FIELDS(#TRANS) FROM\_FILE(TRNTYP) WITH\_KEY(#TRANTYPE) KEEP\_LAST(20) UPRINT FIELDS(#TRANS) ENDSELECT

If the TRANS file contained 3 different state codes and 5 different transaction types then in the case where 10,000 TRANS records were processed the RDML program would do (10,000 + 3 + 5) = 10,008 database accesses. This is a net saving of 19,992 I/Os. This version of the program would probably run in about 1/3 of the time of the original version.

Refer to the FETCH command in this guide for more details of the KEEP\_LAST parameter.

1 5.3 Sample RDML Programs

#### 5.3.10 Generalized Subroutine

The "demand" subroutines described in the previous examples can be generalized and used for any file.

Consider a "generalized" FETCH command like this:

FETCH FIELDS(#FIELD1 #FIELD2 ... #FIELDn) FROM FILE(FILE) WITH KEY(#KEY1 ... #KEYn)

This can be replaced by executing a "demand" subroutine that exactly emulates the FETCH command like this:

EXECUTE SUBROUTINE(GET\_FILE) WITH\_PARMS(#KEY1 ... #KEYn)

The actual "demand" subroutine in generalized format would look like this: SUBROUTINE NAME(GET\_FILE) PARMS((#GETKEY1 \*RECEIVED) "

(#GETKEYn \*RECEIVED))

DEFINE FIELD(#GETKEY1) REFFLD(#KEY1) 11 II II

"

DEFINE FIELD(#GETKEYn) REFFLD(#KEYn)

DEF LIST NAME(#FILE) FIELDS(#KEY1 ... #KEYn #FIELD1 ... #FIELDn) TYPE(\*WORKING) ENTRYS(as required)

LOC ENTRY IN LIST(#FILE) WHERE('(#KEY1 = #GETKEY1) \*AND ...

(#KEYn = #GETKEYn)')

IF\_STATUS IS\_NOT(\*OKAY) FETCH FIELDS(#FIELD1 #FIELD2 ... #FIELDn) FROM\_FILE(FILE) WITH KEY(#GETKEY1 ... #GETKEYn) ADD ENTRY TO LIST(#FILE) **ENDIF** 

**Points to Note:** 

• This routine does not account for 2 possibilities.

- The first is that the working list may overflow. This could be solved by using the COUNTER parameter on the DEF\_LIST command. Before the ADD\_ENTRY command the value could be checked and the list cleared (CLR\_LIST command) if greater than or equal to the ENTRYS value.
- The second is the fact that the requested record may not be found in either the working list or the database. This could be solved by an IF\_STATUS after the FETCH command. If found the record would be added to the working list. If not found appropriate action could be taken (e.g.: ABORT command).
- 1 5.3 Sample RDML Programs

## 5.3.11 Page At a Time Scrolling (Forward Only)

**Initial Example** 

More Efficient Example

### **Files Involved**

Physical file PSLMST (personnel master file) has the following fields and an additional logical view called PSLMSTV1 keyed by SURNAME:

Field	Туре	Length	Description
EMPNO	S	5	Employee number (primary key)
GIVENAME	А	20	Given name
SURNAME	А	20	Surname
ADDRESS1	А	30	Address line 1
ADDRESS2	А	30	Address line 2
ADDRESS3	А	30	Address line 3

1 5.3 Sample RDML Programs

## **Initial Example**

This program performs generic 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 to search for
END LOOP
```

This program will work just fine, but what if the user inputs a search name of "D", and 800 employees working for the company have a surname that starts with "D"?

The result will be a list containing 800 names. But more importantly, it will take a long time to build up the list and use a lot of computer resource while doing it.

To solve this problem, a technique called "page at a time" browsing is often used. What this basically means is that the program extracts one "page" of names matching the request, and then displays them to the user. If the user presses the roll up key then the next page is fetched and displayed, etc, etc.

↑ 5.3.11 Page At a Time Scrolling (Forward Only)

## **More Efficient Example**

This 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
-> DEFINE
            FIELD(#L1POS) TYPE(*CHAR) LENGTH(7)
 DEF_LIST NAME(#L1) FIELDS((#SURNAME) (#GIVENAME) (#EMF
>
       (#ADDR ESS1)) COUNTER(#L1COUNT) PAGE_SIZE(#L1PAGE)
       TOP_ENTRY(# 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$KEY) TO(UP)
-> CHANGE
             FIELD(#L1TOP) TO(1)
-> SELECT
            FIELDS(#L1) FROM FILE(PSLMSTV1)
       WITH KEY(#SURNAME) GENERIC(*YES)
        WHERE('#IO$KEY = UP') OPTIONS(*ENDWHERE)
->
-> EXECUTE SUBROUTINE(DISPLAY) WITH PARMS("'More...")
  ADD_ENTRY TO_LIST(#L1)
  ENDSELECT
  ******
           If names found, display list to user
        COND('#L1COUNT *GT 0')
  IF
-> EXECUTE SUBROUTINE(DISPLAY) WITH_PARMS("Bottom")
  ******
           else issue error indicating none found
  ELSE
             MSGTXT('No employees have a surname matching
  MESSAGE
        request')
  ENDIF
  ******
           Loop back and request next name to search for
```

```
END_LOOP
```

\*\*\*\*\*\*

\*\*\*\*\*\* Display names if page is full or list is complete

\*\*\*\*\*\*

-> SUBROUTINE NAME(DISPLAY) PARMS(#L1POS)

```
> DEFINE FIELD(#L1REMN) TYPE(*DEC) LENGTH(5) DECIMALS(5
```

- -> CHANGE FIELD(#L1REMN) TO('#L1COUNT / #L1PAGE')
- -> IF COND('(#L1COUNT \*NE 0) \*AND (#IO\$KEY = UP) \*AND ((#L1POS = "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 to be displayed and can considerably improve performance in most of them.

It is easy to modify existing programs that use SELECT and DISPLAY (like the initial example here) to use the page at a time technique. Note how the new logic "slots into" the existing logic with no major structural change to the program logic/flow.

The easiest way to implement "page at a time" techniques is to design and fully test a standard "algorithm" that is suitable for your site's needs. This can then be used as a base or template for all future applications.

↑ 5.3.11 Page At a Time Scrolling (Forward Only)

## 5.3.12 Page at a Time Scrolling (Backwards & Forwards)

### **Files Involved**

Physical file DEMNAME (demonstration name and address file) keyed by DEMNAC and an associated logical view called DEMNAMEN (keyed by DEMNAM).

Field	Type	Length	Description
DEMNAC	А	7	Name and address code
DEMNAM	А	25	Full name
DEMAD1	А	25	Address line 1
DEMAD2	А	25	Address line 2
DEMAD3	А	25	Address line 3
DEMPCD	А	4	Post/zip code

## **RDML Program / Subroutine**

The following program is intended to act as a "search" subroutine for any caller program.

It is not intended that it be invoked directly by a user. Any caller program that asks the user to specify a name and address "code" can call this subroutine to allow the user to search through the name and address file by customer name and select the one required.

The selected customer "code" and name are then exchanged back into the caller program's fields for subsequent processing.

The particularly useful thing about this program is that not only can it roll forwards a "page at a time", it can roll backwards a "page at a time" (even past the original starting point), and it allows the user to "jump around" anywhere in the file at any time.

\*\*\*\*\*\*\*\*\* Define work variables and constants

DEFINE #PAGESIZE REFFLD(#LISTCOUNT) DEFAULT(14) DEFINE #LISTSIZE REFFLD(#LISTCOUNT) DEFINE #WORKSIZE REFFLD(#LISTCOUNT)

DEFINE FIELD(#SEARCHNAM) REFFLD(#DEMNAM) LABEL('Search for')

DEFINE FIELD(#TOPPAGNAM) REFFLD(#DEMNAM) DEFAULT(X'FF

DEFINE FIELD(#BOTPAGNAM) REFFLD(#DEMNAM) DEFAULT(\*BL

**\*\*\*\*\*\*** Define identical list to display and list to work with

DEF\_LIST NAME(#LIST01) FIELDS((#SELECTOR \*SELECT)(#DEMNAC)(#DEMNAM)(#DEMAD1)(#DEMPCD)) COUNTER(#LISTSIZE)

DEF\_LIST NAME(#WORK01) FIELDS((#DEMNAC)(#DEMNAM)(#DEMAD1)(#DEMPCD)) COUNTER(#WORKSIZE) TYPE(\*WORKING) ENTRYS(14)

\*\*\*\*\*\*\*\* Define a permanent exchange list and file open options

EXCHANGE FIELDS(#DEMNAC #DEMNAM) OPTION(\*ALWAYS) OPEN FILE(DEMNAMEN) USE\_OPTION(\*ONDEMAND)

\*\*\*\*\*\*\*\*\* Process search requests until a name is selected or \*\*\*\*\*\*\*\*\* function key 24 is used to end the program

**BEGIN\_LOOP** 

REQUEST FIELDS((#SEARCHNAM)) BROWSELIST(#LIST01) EXIT\_KEY(\*NO) MENU\_KEY(\*NO) USER\_KEYS((\*ROLLUP 'Up')(\*ROLLDOWN 'Down') (24 'End'))

CASE OF\_FIELD(#IO\$KEY)

WHEN ('= UP') /\* Roll up key \*/ EXECUTE ROLL (UP #BOTPAGNAM)

WHEN ('= DN') /\* Roll down key \*/ EXECUTE ROLL (DOWN #TOPPAGNAM)

WHEN ('= "24'") /\* Fnc key 24 \*/ CHANGE FIELD(#DEMNAC) TO("'?"')

#### CHANGE FIELD(#DEMNAM) TO(\*BLANKS) RETURN

OTHERWISE /\* Enter key \*/ SELECTLIST #LIST01 GET\_ENTRYS(\*SELECT) RETURN ENDSELECT EXECUTE ROLL (UP #SEARCHNAM)

ENDCASE

END\_LOOP

*****	ROLL : Roll page backwards/forwards
*****	DIRECTION : Direction to roll (UP/DN)
*****	STARTNAM : Name at which to start roll

#### SUBROUTINE ROLL PARMS((#DIRECTION \*RECEIVED) (#STARTNAM \*RECEIVED))

DEFINE FIELD(#DIRECTION) TYPE(\*CHAR) LENGTH(4) DEFINE FIELD(#STARTNAM) REFFLD(#DEMNAM)

CLR\_LIST NAMED(#WORK01)

\*\*\*\*\*\*\* Handle a roll up request

IF COND('#DIRECTION = UP')
SELECT FIELDS((#WORK01)) FROM\_FILE(DEMNAMEN)
WHERE('#WORKSIZE \*LT #PAGESIZE')
WITH\_KEY(#STARTNAM) OPTIONS(\*STARTKEY
\*ENDWHERE)
ADD\_ENTRY TO\_LIST(#WORK01)
ENDSELECT

\*\*\*\*\*\*\* Handle a roll down request

ELSE SELECT FIELDS((#WORK01)) FROM\_FILE(DEMNAMEN)

```
WHERE('#WORKSIZE *LT #PAGESIZE')
WITH_KEY(#STARTNAM)
OPTIONS(*STARTKEY *ENDWHERE *BACKWARDS)
ADD_ENTRY TO_LIST(#WORK01)
ENDSELECT
SORT_LIST NAMED(#WORK01) BY_FIELDS((#DEMNAM)) /* Important
ENDIF
```

\*\*\*\*\*\*\* Map work list to browse list for display

CLR\_LIST NAMED(#LIST01) SELECTLIST NAMED(#WORK01) ADD\_ENTRY TO\_LIST(#LIST01) ENDSELECT

\*\*\*\*\*\*\*\* Set/save pointers to top and bottom of displayed page

IF COND('#WORKSIZE = 0')

CHANGE FIELD(#TOPPAGNAM) TO(X'FF')

CHANGE FIELD(#BOTPAGNAM) TO(\*BLANKS)

MESSAGE MSGTXT('Search request is beyond start or end of names file')

ELSE

GET\_ENTRY NUMBER(1) FROM\_LIST(#WORK01) CHANGE FIELD(#TOPPAGNAM) TO(#DEMNAM) GET\_ENTRY NUMBER(#WORKSIZE) FROM\_LIST(#WORK01) CHANGE FIELD(#BOTPAGNAM) TO(#DEMNAM) ENDIF

#### ENDROUTINE

The type of RDML program that calls this program would probably include code like this:

\*\*\*\*\*\*\* Ask user to input a customer name code

REQUEST FIELDS((#DEMNAC) .... etc, etc)

\*\*\*\*\*\*\* Validate name code (with optional search allowed)

```
BEGINCHECK

IF COND('#DEMNAC = "?"")

CALL PROCESS(XXXXXXXX) FUNCTION(YYYYYY)

ENDIF

FILECHECK FIELD(#DEMNAC) WITH_FILE(DEMNAME)

MSGTXT('No customer with name & address

code exists')

ENDCHECK
```

If the user enters a name and address code, it is immediately validated against file DEMNAME. If it is valid, the program continues. If it is not, control is returned to the REQUEST command and the error message appears.

If however, the user enters a "?" as the name and address code the previously described function is called. It allows the user to search the name and address file.

If the user selects a name the associated DEMNAC value is returned into this program, thus the FILECHECK will work.

If the user does not select a name (i.e.: uses function key 24 to end the search) the DEMNAC value returned into this program is a "?", which will cause the FILECHECK command to fail to find a record, thus triggering a validation error.

1 5.3 Sample RDML Programs

## **5.4 Producing Reports Using LANSA**

If the UPRINT (unformatted print) command is ignored, then reports are produced by LANSA with the following commands:

DEF_HEAD	Define report heading line(s)
DEF_FOOT	Define report footing line(s)
DEF_LINE	Define report detail line(s)
DEF_BREAK	Define report break line(s)
PRINT	Print detail line(s) or trigger break line(s)
SKIP	Skip to a specific line on the report
SPACE	Space a number of lines on the report

The first 4 commands are used to define the various components of a report layout and the last 3 to actually control report production.

Each of these commands is described in detail in the *RDML Commands* chapter of the Visual LANSA Technical Reference Guide.

The report production commands have been designed this way in an attempt to satisfy 2 differing requirements:

- 1. To be "smart" enough to satisfy most common reporting requirements virtually automatically.
- 2. To provide sufficient flexibility to allow complex reporting requirements to be satisfied.

Most common reporting requirements can be satisfied by using the DEF\_HEAD, DEF\_BREAK, DEF\_FOOT, DEF\_LINE and PRINT commands in conjunction with a very small amount of RDML program logic.

When more complex reports are involved, using the DEF\_LINE, PRINT, SPACE and SKIP commands in conjunction with more RDML program logic will probably produce the desired result.

The examples in this appendix are mainly involved with the first case. That is, producing common reports by utilizing the "smart" features of the report production commands.

The following topics describe the simplest type of report that can be produced and then successively build on this example to demonstrate how each of the report production commands are used to produce complex reports.

The following topics use examples to explain how reports can be produced by LANSA:

- 5.4.1 Using the DEF\_LINE Command
- 5.4.2 Using the DEF\_HEAD Command
- 5.4.3 Numbering Report Pages
- 5.4.4 Using the DEF\_FOOT Command
- 5.4.5 Using a Leading DEF\_BREAK Command
- 5.4.6 Using a Trailing DEF\_BREAK Command
- 5.4.7 Printing on Multiple Lines
- 5.4.8 Submitting Reports to Batch

Finally, some of the most common problems that occur when first using LANSA to produce reports, in 5.4.9 Reporting Q & A.

#### Also See

FRM115 - Writing Reports exercise in the LANSA Fundamentals Windows *Tutorials*.

### 5.4.1 Using the DEF\_LINE Command

All the following examples of report production use a file called ACCOUNTS that contains the following records. It is assumed that the file is ordered (i.e.: keyed) by company, division and department.

Company (#COMP)	Division (#DIV)	Department (#DEPT)	Expenditure (#EXPEND)	Revenue (#REVNU)
01	1	ADM	400	576
"	"	MKT	678	56
"	"	SAL	123	6784
"	2	ADM	46	52
"	"	SAL	978	456
"	3	ACC	456	678
"		SAL	123	679
02	1	ACC	843	400
"	"	MKT	23	0
"		SAL	876	10
"	2	ACC	0	43

**Note** that although all the examples deal with one file only, for the sake of simplicity, there is no restriction on the number of files that can be accessed by or printed by an RDML program.

The simplest type of report that can be produced is a straight listing from all or

part of a file.

To list all of the ACCOUNTS file, all that is required is:

DEF\_LINE NAME(#ACCOUNT) FIELDS(#COMP #DIV #DEPT #EXPEND #REVNU) SELECT FIELDS(#ACCOUNT) FROM\_FILE(ACCOUNTS) PRINT LINE(#ACCOUNT) ENDSELECT ENDPRINT

This would produce a report that looked something like this:

Company	Division	Department	Expenditure	Revenue
01 1	ADM	400	576	
01 1	MKT	678	56	
01 1	SAL	123	6784	
01 2	ADM	46	52	
01 2	SAL	978	456	
01 3	ACC	456	678	
01 3	SAL	123	679	
02 1	ACC	843	400	
02 1	MKT	23	0	
02 1	SAL	876	10	
02 2	ACC	0	43	

Of course, this example can be changed in various ways to only print part of the file - such as only the records for a selected company:

DEF\_LINE NAME(#ACCOUNT) FIELDS(#COMP #DIV #DEPT #EXPEND #REVNU) REQUEST FIELDS(#COMP) SELECT FIELDS(#ACCOUNT) FROM\_FILE(ACCOUNTS) WITH\_KEY(#COMP) PRINT LINE(#ACCOUNT) ENDSELECT ENDPRINT

Or, only records where the difference between expenditure and revenue is less than a specified amount:

```
DEF_LINE NAME(#ACCOUNT) FIELDS(#COMP #DIV
#DEPT #EXPEND #REVNU)
DEFINE FIELD(#MAXLOSS) REFFLD(#REVNU)
LABEL('Allowable loss')
REQUEST FIELDS(#MAXLOSS)
SELECT FIELDS(#ACCOUNT) FROM_FILE(ACCOUNTS)
WHERE('(#REVNU - #EXPEND) *LT #MAXLOSS')
PRINT LINE(#ACCOUNT)
ENDSELECT
ENDPRINT
```

This final example uses the \*ONCHANGE field attribute to only print the company number when it changes, rather than on every line:

DEF\_LINE NAME(#ACCOUNT) FIELDS((#COMP \*ONCHANGE) #DIV #DEPT #EXPEND #REVNU) SELECT FIELDS(#ACCOUNT) FROM\_FILE(ACCOUNTS) PRINT LINE(#ACCOUNT) ENDSELECT ENDPRINT

This would produce a report that looks something like this:

Company		Division D	epartment	Expenditure	Revenue
01	1	ADM	400	576	
	1	MKT	678	56	
	1	SAL	123	6784	
	2	ADM	46	52	
	2	SAL	978	456	
	3	ACC	456	678	
	3	SAL	123	679	
02	1	ACC	843	400	
	1	MKT	23	0	
	1	SAL	876	10	
	2	ACC	0	43	

↑ 5.4 Producing Reports Using LANSA

### **5.4.2 Using the DEF\_HEAD Command**

The example described in 5.4.1 Using the DEF\_LINE Command can be altered to place the company number into the report heading and start a new page on change of company number like this:

GROUP\_BY NAME(#ACCOUNT) FIELDS(#COMP #DIV #DEPT #EXPEND #REVNU) DEF\_HEAD NAME(#HEADING) FIELDS(#COMP) TRIGGER\_BY(#COMP) DEF\_LINE NAME(#DETAILS) FIELDS(#DIV #DEPT #EXPEND #REVNU) SELECT FIELDS(#ACCOUNT) FROM FILE(ACCOUNTS

SELECT FIELDS(#ACCOUNT) FROM\_FILE(ACCOUNTS) PRINT LINE(#DETAILS) ENDSELECT ENDPRINT

This would now produce a 2 page report that looked something like this:

Company 01

Division	Department	Expe	enditure	Revenue
1	ADM	400	576	
1	MKT	678	56	
1	SAL	123	6784	
2	ADM	46	52	
2	SAL	978	456	
3	ACC	456	678	
3	SAL	123	679	

Company 02

Division	Department	Expenditure		Revenue
1	ACC	843	400	
1	MKT	23	0	
1	SAL	876	10	

2 ACC 0 43

Note how the DEF\_HEAD command indicated that field #COMP was to be printed in the report heading area and the heading was to be "triggered" by change of #COMP (i.e.: a new page started when the company number changes).

In fact, the TRIGGER\_BY parameter may be strictly incorrect in this situation. If company 01 had more records that would fit on 1 page the report would look something like this:

Company 01		Page 1		
Division	Department	Expe	nditure	Revenue
1	ADM	400	576	
1	MKT	678	56	
1	SAL	123	6784	
2	ADM	46	52	
2	SAL	978	456	
3	ACC	456	678	
3	SAL	123	679	

Page 2

Division	Department	Expe	nditure	Revenue
4	ADM	400	576	
4	MKT	678	56	
4	SAL	123	6784	
5	ADM	46	52	
5	SAL	978	456	
5	ACC	456	678	
5	SAL	123	679	

Division	Department	Expenditure		Revenue
1	ACC	843	400	
1	MKT	23	0	
1	SAL	876	10	
2	ACC	0	43	

Note how the company number is not printed on page 2. This is because the DEF\_HEAD line is only to be "triggered" by change of company number. Between page 1 and 2 the company number did not change, so the DEF\_HEAD command details are not printed. If however, the DEF\_HEAD command was specified as:

DEF\_HEAD NAME(#HEADING) FIELDS(#COMP) TRIGGER\_BY(#COMP \*OVERFLOW)

which indicates that the DEF\_HEAD command should be triggered by change of company number **or** page overflow, then "Company 01" would also print on the start of page 2.

↑ 5.4 Producing Reports Using LANSA

## **5.4.3 Numbering Report Pages**

LANSA allows up to 8 reports to be produced at any one time. All report definition commands have a FOR\_REPORT parameter associated with them. This parameter defaults to 1 and is used to indicate which report the line definition is associated with. Any number in the range 1 -> 8 can be specified for this parameter.

In addition, each of the 8 allowable reports has a "special" field associated with it called REPnPAGE which can be used to print the page number on the report. The "n" indicates the report number with which the page number field is associated.

So, to expand the previous example to include a page number, all that is required is the following:

```
GROUP_BY NAME(#ACCOUNT) FIELDS(#COMP #DIV #DEPT
#EXPEND #REVNU)
```

- DEF\_HEAD NAME(#HEADING) FIELDS(#COMP #REP1PAGE) TRIGGER\_BY(#COMP \*OVERFLOW)
- DEF\_LINE NAME(#DETAILS) FIELDS(#DIV #DEPT #EXPEND #REVNU)

SELECT FIELDS(#ACCOUNT) FROM\_FILE(ACCOUNTS) PRINT LINE(#DETAILS) ENDSELECT ENDPRINT

This would now produce a two page report that looked something like the examples following.

Company 01 Page 1

Division	Department	Expe	enditure	Revenue
1	ADM	400	576	
1	MKT	678	56	
1	SAL	123	6784	
2	ADM	46	52	
2	SAL	978	456	
3	ACC	456	678	

3 SAL 123 679

Company 02 Page 2

Division	Department	Expe	nditure	Revenue
1	ACC	843	400	
1	MKT	23	0	
1	SAL	876	10	
2	ACC	0	43	

### **5.4.4 Using the DEF\_FOOT Command**

The example used in 5.4.2 Using the DEF\_HEAD Command could be expanded still further to include a page number and some text on the report foot line like this:

```
GROUP_BY NAME(#ACCOUNT) FIELDS(#COMP #DIV #DEPT
#EXPEND #REVNU)
DEF_HEAD NAME(#HEADING) FIELDS(#COMP #REP1PAGE)
TRIGGER_BY(#COMP *OVERFLOW)
DEF_LINE NAME(#DETAILS) FIELDS(#DIV #DEPT #EXPEND #REVN
DEF_FOOT NAME(#FOOTING) FIELDS(#REP1PAGE)
TEXT(('EXP/REV REPORT' 58 10))
SELECT FIELDS(#ACCOUNT) FROM_FILE(ACCOUNTS)
PRINT LINE(#DETAILS)
ENDSELECT
ENDPRINT
```

This would now produce a 2 page report that looked something like these examples:

Company 01 Page 1

Division	Department	Expe	nditure	Revenue
1	ADM	400	576	
1	MKT	678	56	
1	SAL	123	6784	
2	ADM	46	52	
2	SAL	978	456	
3	ACC	456	678	
3	SAL	123	679	

Page 1 EXP/REV REPORT

Company 02 Page 2

Division	Department	Expe	nditure	Revenue
1	ACC	843	400	
1	MKT	23	0	
1	SAL	876	10	
2	ACC	0	43	

Page 2 EXP/REV REPORT

1 5.4 Producing Reports Using LANSA

### 5.4.5 Using a Leading DEF\_BREAK Command

The original example RDML program that printed the ACCOUNTS file with the company number in the page heading looked like this:

GROUP\_BY NAME(#ACCOUNT) FIELDS(#COMP #DIV #DEPT #EXPEND #REVNU) DEF\_HEAD NAME(#HEADING) FIELDS(#COMP) TRIGGER\_BY(#COMP \*OVERFLOW) DEF\_LINE NAME(#DETAILS) FIELDS(#DIV #DEPT #EXPEND #REVNU)

SELECT FIELDS(#ACCOUNT) FROM\_FILE(ACCOUNTS) PRINT LINE(#DETAILS) ENDSELECT ENDPRINT

which produced a 2 page report like this:

Company 01

Division	Department	Expe	nditure	Revenue
1	ADM	400	576	
1	MKT	678	56	
1	SAL	123	6784	
2	ADM	46	52	
2	SAL	978	456	
3	ACC	456	678	
3	SAL	123	679	

### Company 02

Division	Department	Exp	enditure	Revenue
1	ACC	843	400	
1	MKT	23	0	
1	SAL	876	10	

```
2 ACC 0 43
```

The layout of this report can be easily improved by using a "leading" break line to "cluster" the divisions together. This could be done like this:

```
GROUP_BY NAME(#ACCOUNT)

FIELDS(#COMP #DIV #DEPT #EXPEND #REVNU)

DEF_HEAD NAME(#HEADING) FIELDS(#COMP)

TRIGGER_BY(#COMP *OVERFLOW)

DEF_BREAK NAME(#BREAK01) FIELDS(#DIV)

TRIGGER_BY(#COMP #DIV) TYPE(*LEADING)

DEF_LINE NAME(#DETAILS) FIELDS(#DEPT #EXPEND #REVNU)

SELECT FIELDS(#ACCOUNT) FROM_FILE(ACCOUNTS)

PRINT LINE(#DETAILS)

ENDSELECT
```

ENDPRINT

Note how the DEF\_BREAK command indicates the field #DIV is to be printed and it is to be "triggered" by change of #COMP **or** change of #DIV. The resulting report would probably look something like this:

Company 01

Department	Expenditure		Revenue
Division 1			
ADM	400	57	6
MKT	678	56	
SAL	123	6784	1
Division 2			
ADM	46	52	
SAL	978	456	
Division 3			
ACC	456	678	}
SAL	123	679	

# Company 02

Department	Expend	iture	Revenue
Division 1			
ACC	843	400	)
MKT	23	0	
SAL	876	10	
Division 2			
ACC	0	43	

## 5.4.6 Using a Trailing DEF\_BREAK Command

"Trailing" break lines are typically used to insert level or break totals. As an example of this, consider the following modifications to the RDML program used in the 5.4.5 Using a Leading DEF\_BREAK Command to demonstrate "leading" break lines:

```
GROUP BY NAME(#ACCOUNT)
     FIELDS(#COMP #DIV #DEPT #EXPEND #REVNU)
DEF HEAD NAME(#HEADING) FIELDS(#COMP)
     TRIGGER_BY(#COMP *OVERFLOW)
DEF BREAK NAME(#BREAK01) FIELDS(#DIV)
     TRIGGER BY(#COMP #DIV)
     TYPE(*LEADING)
DEF BREAK NAME(#BREAK02) FIELDS(#DIV #DIVEXP #DIVRNU)
     TRIGGER BY(#COMP #DIV) TYPE(*TRAILING)
DEF LINE NAME(#DETAILS) FIELDS(#DEPT #EXPEND #REVNU)
DEFINE
         FIELD(#DIVEXP) REFFLD(#EXPEND) LABEL('Expend')
DEFINE
         FIELD(#DIVRNU) REFFLD(#REVNU) LABEL('Revenue')
SELECT
         FIELDS(#ACCOUNT) FROM FILE(ACCOUNTS)
KEEP_TOTAL OF_FIELD(#EXPEND) IN_FIELD(#DIVEXP)
     BY FIELD(#COMP #DIV)
KEEP TOTAL OF FIELD(#REVNU) IN FIELD(#DIVRNU)
     BY FIELD(#COMP #DIV)
PRINT
        LINE(#DETAILS)
ENDSELECT
ENDPRINT
```

Note how the DEF\_BREAK command is "triggered" by change of #COMP or #DIV. Note also the KEEP\_TOTAL commands into the SELECT loop. These are used to automatically keep the company/division subtotals. Refer to the KEEP\_XXXXX command series for more details of these commands and their usage.

The resulting report, using **default design attributes**, would probably look something like this:

Company 01

Department Expenditure Revenue Division 1 ADM 400 576 MKT 678 56 6784 SAL 123 Division 1 Expend 1201 Revenue 7416 Division 2 ADM 46 52 SAL 978 456 Division 2 Expend 1024 Revenue 508 Division 3 ACC 456 678 679 SAL 123 Division 3 Expend 579 Revenue 1357

Company 02

Department Expenditure Revenue Division 1 ACC 843 400 MKT 23 0 SAL 876 10 Division 1 Expend 1742 Revenue 410 Division 2 ACC 0 43 Division 2 Expend 0 Revenue 43

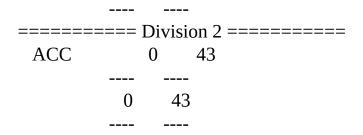
However, by using the Report Design Facility in the *LANSA for i* User Guide, the entire report layout could be modified in a few minutes to look like the examples following.

### Company 01

Department Expenditure Revenue ======= Division 1 ======== ADM 400 576 MKT 678 56 SAL 123 6784 \_\_\_\_ \_\_\_\_ 1201 7416 \_\_\_\_ \_\_\_\_ ======= Division 2 ======== ADM 46 52 978 SAL 456 \_\_\_\_ \_\_\_\_ 1024 508 \_\_\_\_ \_\_\_\_ ======= Division 3 ========= ACC 456 678 123 SAL 679 \_\_\_\_ 579 1357 \_\_\_\_ \_\_\_\_

### Company 02

Department Expenditure Revenue



1 5.4 Producing Reports Using LANSA

### **5.4.7 Printing on Multiple Lines**

All DEF\_XXXXX commands allow the definition of print lines that span multiple print lines. For instance, to print a report heading like this:

you can use three separate DEF\_HEAD commands like this:

DEF\_HEAD NAME(#HEAD01) FIELDS((#DATE \*L1 \*C2)) DEF\_HEAD NAME(#HEAD02) FIELDS((#COMP \*L2 \*C2)) DEF\_HEAD NAME(#HEAD03) FIELDS((#DIV \*L3 \*C2))

or just one DEF\_HEAD command like this: DEF\_HEAD NAME(#HEAD) FIELDS((#DATE \*L1 \*C2) (#COMP \*L2 \*C2) (#DIV \*L3 \*C2))

Of these two methods, the last one is preferred because:

- It is slightly more efficient, both during function definition and during function execution.
- It is easier to use with the report design facility.

When setting up heading (DEF\_HEAD) or foot (DEF\_FOOT) lines you should use one and only one command to define all print lines where ever possible. However, when setting up detail (DEF\_LINE) or break (DEF\_BREAK) lines there is one more consideration to be made in deciding whether to use one DEF\_XXXXX command or more.

Consider a detail report that is to look like this:

### 

### 

These detail lines could be defined like this (row and column details have been omitted):

DEF\_LINE NAME(#DET01) FIELDS(#CUSNAM #CUSAD1) IDENTIFY(\*COLHDG) DEF\_LINE NAME(#DET02) FIELDS(#CUSAD2) IDENTIFY(\*NOID) DEF\_LINE NAME(#DET03) FIELDS(#CUSAD3) IDENTIFY(\*NOID) DEF\_LINE NAME(#DET04) FIELDS(#POSTCD) IDENTIFY(\*LABEL)

or like this:

```
DEF_LINE NAME(#DETAIL) FIELDS((#CUSNAM *L1 *C2 *COLHDG)
(#CUSAD1 *L1 *C39 *COLHDG)
(#CUSAD2 *L2 *C39 *NOID )
(#CUSAD3 *L3 *C39 *NOID )
(#POSTCD *L4 *C39 *LABEL ))
```

These two methods of defining the detail portion of the report may produce slightly different results.

Since all the lines defined in one DEF\_XXXXX command are considered to be part of the same "cluster" of lines they are always printed on the same page.

Thus, by using the first definition method (four separate commands) information about a customer may be "split" over two pages like this:

### 

#### 

### 

If the second method was used (one command to define all four print lines), this would not happen. If all four print lines could not be fitted onto the page a new page would have been started.

The choice here depends upon whether or not the "split" is acceptable or even required for the report being designed.

↑ 5.4 Producing Reports Using LANSA

# **5.4.8 Submitting Reports to Batch**

One of the most common techniques used in IBM i applications when producing "on demand" reports for end users is to actually write 2 programs.

The **first** program is run interactively and requests that the user specify the criteria that are to be used when producing the report. A call to the second program is then submitted to batch, passing the user's criteria as parameters.

The **second** program, when it is invoked in batch, uses the criteria received as parameters to produce the required report.

The main reason that this technique is used is that most reporting programs perform too many database I/Os to be efficiently run in an interactive environment, thus they are submitted to batch.

However, it is possible in LANSA to achieve exactly the same result using only one program. As an example, consider the following function, PRNTBCH, which belongs to process GLPROC01:

```
DEF_LINE NAME(#LINE01)
FIELDS(#GLNUMB #BATCH #CREDIT #DEBIT)
DEFINE FIELD(#MINCREDIT) REFFLD(#CREDIT)
```

```
IF COND('*JOBMODE = I')

REQUEST FIELDS(#GLNUMB #BATCH #MINCREDIT)

SUBMIT PROCESS(GLPROC01) FUNCTION(PRNTBCH)

EXCHANGE(#GLNUMB #BATCH #MINCREDIT)

ELSE

SELECT FIELDS(#CREDIT #DEBIT) FROM_FILE(GLMASTV3)

WITH_KEY(#GLNUMB #BATCH)

WHERE('#CREDIT *GE #MINCREDIT')

PRINT LINE(#LINE01)

ENDSELECT

ENDPRINT
```

### ENDIF

When run **interactively** (i.e.: \*JOBMODE = I), this function requests that the user input a general ledger number, a batch number and a minimum credit amount. It then submits **itself** to batch, exchanging the general ledger number, batch number and minimum credit amount that were input by the user. When invoked in **batch**, this function selects only records that match the user's request from logical file GLMASTV3 and prints them. The initial value of fields #GLNUMB, #BATCH and #MINCREDIT are established from the exchange list that was passed to the batch version of this function from the interactive version.

↑ 5.4 Producing Reports Using LANSA

# 5.4.9 Reporting Q & A

The following list describes some of the most common problems that occur when first using LANSA to produce reports and their causes.

## Why do fields come out as blanks or zeros?

The most common cause is that the field has been referenced in line definition commands, but is not actually being retrieved from the database. Check all database read commands such as SELECT, FETCH, etc to make sure that the field is in fact being retrieved from the database.

## What causes level breaks to appear randomly or out of order?

The most likely reason is that the data being provided to the print command is not in the expected order. Remember that LANSA does not sort the data automatically. The data must be accessed via a file that presents it to the RDML program in the required order. Check the keys of the file being used to ensure the data is being retrieved in the expected order, then check that all "trigger" parameters are consistent with the processing order.

# What causes my last set of level breaks or grand total to be missing?

The most common cause is a missing ENDPRINT command. The last set of level break lines (including "grand totals") must be "flushed" to the report by executing an ENDPRINT command. If no specific ENDPRINT command is coded, it will be automatically issued when the function ends. This method of automatically issuing an ENDPRINT may not always achieve the desired results.

↑ 5.4 Producing Reports Using LANSA

# 6. Creating Applications Using Components

With LANSA components, you can create applications with powerful graphical user interfaces (GUI) using LANSA's repository-based development method.

With components you create event-driven applications using an advanced visual designer together with an easy-to-use code editor. You can accomplish most development tasks graphically.

Applications are assembled from components with extensive built-in functionality. Because Visual LANSA is based on component technology and repository approach, you can reuse every part of an application. You can store groups of fields, menus, pictures, and other parts of an application with the associated code in the LANSA repository and reuse them. The reuse of these standard elements in all applications ensures rapid development, consistency and ease of maintenance.

The language you use in creating Visual LANSA components is the familiar RDML with a few extensions required for event-driven programming (RDMLX). The event-driven programming method provides the foundation for easy-to-use applications which put the user in control. However, new commands have been kept to a minimum to ensure that you can use a single set of skills for Windows, IBM i and web application development.

### Also see

6.1 You Work in a Graphical Editor

6.2 You Create Powerful Graphical Interfaces

6.3 You Use a Repository-Based Graphical Designer

6.4 The Component Model

6.5 RDMLX Is Extended RDML

6.6 Move Existing Applications to Visual LANSA

6.7 A Few Things You Need to Know

If you're unfamiliar with this topic, you should review some or all of the following:

6.8 Component Basics	6.17 Menus	6.26 Property Sheets
6.9 Event-Driven	6.18 Displaying	6.27 Keyboard and Mouse
Programs	Messages and Errors	Components
6.10 Work with	6.19 Multi-Form	6.28 Drag-and-Drop

Components
6.11 Create an
Application
6.12 Enrolling Bitmaps
and Icons
6.13 Forms
6.14 Controls
6.15 Lists, Tree Views, Grids and Graphs
6.16 Tab Folder Framework

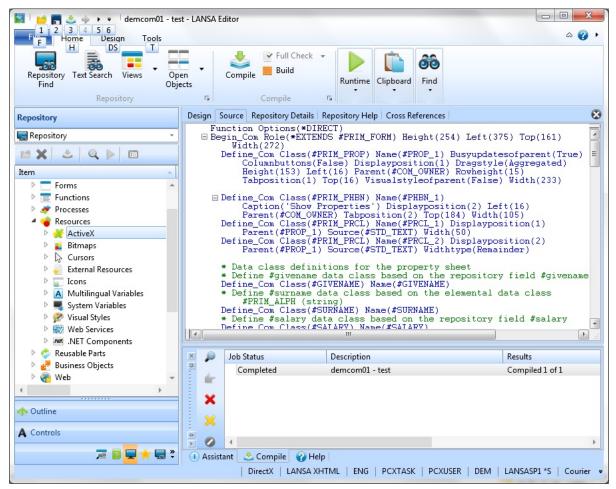
Applications 6.20 Reusable Parts 6.21 Ancestor Components 6.22 Collections 6.23 Referencing 6.24 Data Classes 6.25 Using Common Dialogs 6.29 What the Application will look like
6.30 Remember the User's Choices
6.31 Help Text and Documenting Components
6.32 GUI Guidelines
6.33 Converting an Existing Application

↑ Visual LANSA Developer Guide

## 6.1 You Work in a Graphical Editor

You create the interface of your components graphically by dragging controls and other components from the repository to a form. The form represents a window of the application when it is running. This means that you can see what your application will look like while you are developing it.

Working in a graphical environment can significantly shorten the time you spend creating an application. This is because you create much of the application using drag-and-drop so there is less code to write.



To understand what a difference working in the graphical development environment makes, try it out. Even if you do not have a Visual LANSA hardware key, you can create components and have a look at the Visual LANSA work environment. To do this, create a new form called DEMCOM01. (You can create ten forms or other components named DEMCOM01 to DEMCOM10.)

# 

# 6.2 You Create Powerful Graphical Interfaces

The graphical components used to build applications provide you with powerful ways of displaying, manipulating and navigating data. They offer a wide variety of ways of communicating with the user, including explorer-type navigation, graphics, animations, videos and sound. As a result you can create flexible and compact applications which are very easy to use.

To see what you can do in a graphical user interface, run the VL\_DEM20 application shipped with LANSA. (To do this, locate and select VL\_DEM20 form and then choose the Execute command from the pop-up menu or the ribbon.)

mployee Details <u>S</u> kills	: E <u>m</u> ail	
Employee Number	A0070	- Andrews
Employee Surname	BROWN	
Employee Given Name(s)	VERONICA	
Business Phone Number	(02) 9647 2788	A COLOR
Home Phone Number	(02) 9609 4627	-1-/-
Start Date (DDMMYY)	28/01/90	
Termination Date (DDMMYY)	0/00/00	No. of the second se
Street No and Name	12 Railway Street	
Suburb or Town	Baulkham Hills	Enlarge Photo
State and Country	NSW Australia	Save
Post / Zip Code	2153	Cancel
Employee Salary	50,125.00	Delete

When you run the demo application, note:

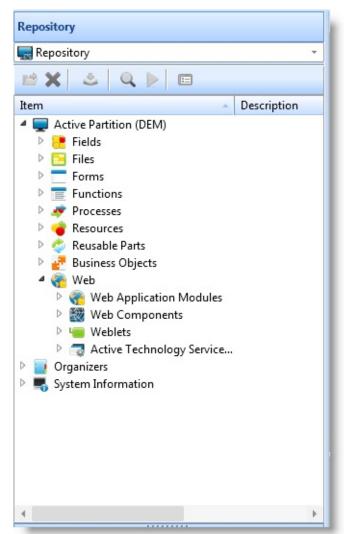
- How easy it is to use. Even if you have never used it before, you don't need any instructions because the interface shows visually what you can do.
- How much information is shown in a single window. You can create very compact applications using components such as tabbed folders which offer powerful ways of navigating and grouping data.
- How much more you know about the employees because of the photograph and the video. Pictures and sound have a high information content.

- How flexible the application is. You can move to any part of the application with a few clicks. You can locate employees by department and section as well as by name. The program does not force you to work with the information in any set order.
- What effective ways of presenting list-type information grids (Skills) and graphs (Department Statistics) offer.
- How you can incorporate other applications in Visual LANSA. Try out sending Email to yourself using the Email tab.
- How the application responds immediately to your actions. Try changing some of the employee details and then moving on without saving the changes.

When you create a graphical user interface, you have lots of options. You can choose from many components, decide how many windows to use, design the layout including colors and fonts, and determine the best way the user should interact with the program. This means that you can customize the interface to the smallest detail.

## 6.3 You Use a Repository-Based Graphical Designer

Unlike most other GUI tools, Visual LANSA implements a repository-based approach. As always, the LANSA repository centralizes the common business information about your applications.



This means that you get all the benefits characteristic of LANSA application development in addition to GUI capabilities:

- A short learning curve
- Productivity
- Reuse
- Simplicity
- Ease-of maintenance

- Consistency
- Multilingual support

A repository approach is essential when developing GUI applications. This is because in this environment applications are quick to develop and as a consequence their number can grow rapidly. Without a repository approach you will easily end up in a situation where you have lots of applications with a lot of duplicated code. This means that the time you have gained in developing the applications is quickly spent maintaining them. The repository gives you the means to control the development process by ensuring consistency and reuse and by making the maintenance of applications quick and simple.

Also, it is quite possible that your applications may lack a common look and feel, no matter how strict interface standards you have created, because GUI development gives you so much freedom. Here again the repository helps you to stay in control by imposing uniformity of visual representation.

# You Define Data in the Repository Once

As an example, let's look at how you would handle an employee number field in a typical GUI application and in LANSA.

In a typical GUI tool (such as Visual Basic):

• You present the employee number as a text field with a separate label.



• You have to create a new text field and a new label for every application and every window where employee number is used.

It's likely that the text, the character case, the font, the size and maybe even the colors of the labels are slightly different in every application, especially if more than one developer is involved. Some developers might add spin buttons to the edit box so that the user can scroll employee numbers. Others might use a combo box instead of a text box.

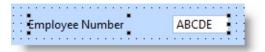
- You have to size the text field to fit the employee number every time it is used. To find out the required size, you have to execute the application to fetch data from the database to see if it fits.
- To retrieve or insert the employee number in the application, you have to explicitly associate the text field with a record in the database.
- You have to code the validation logic for employee number separately in

every application.

- You have to code error handling for employee number separately in every application.
- In a localized version you have to substitute the translated text of the label in a separate resource file for every occurrence of the label.

Whereas in Visual LANSA:

- You store the employee number in the repository as a field with a description (label), validation rules, error handling and help text. You can enter the label in multiple languages. Of course you do this only once.
- When developing an application, you drag the employee number field to any window where this information is required. This is all you need to do.



Note that:

- The field is displayed with a label and it is sized to fit the field in the database automatically.
- When you change the label in the field definition, this change will be reflected in every application where the field is used.
- You do not have to write any validation rules or error handling because they have been defined in the repository.
- The field is implicitly bound to the field in the database. When you execute the application, you can specify in which language the labels are shown.

Think about how you maintain applications in these different environments. For example, what would you need to do if the length of the employee number is changed?

## Intelligent Lists, Grids and Graphs Are Based on the Repository

The power of the advanced list, tree view, grid and graph components in Visual LANSA is also based on the repository. You define the data to be shown in these components simply by dragging fields to them. For instance you create a list of employee numbers by dragging the employee number field to a list component.

## **Visual Styles**

The appearance of Visual LANSA components is controlled by visual style components. A visual style component controls the appearance of the application including colors, fonts and 3D effects.

You can set one visual style to govern your entire application. This ensures absolute uniformity and makes it possible to make global changes by changing a single setting. Visual styles are multilingual so that you can define different settings for different languages in one style.

# 6.4 The Component Model

Visual LANSA extends the LANSA repository to include components in addition to fields and files. This object-oriented component model provides the foundation for user-centered, event-driven applications. However, in Visual LANSA the object-oriented paradigm is implemented in a simple way to allow you to focus on productivity and business goals.

The component model is the foundation for future application development with LANSA.

### **Fields Become Components Automatically**

All your existing LANSA fields become components automatically. By default all fields become entry fields. This means that when you want to use a field in an application, you drag it from the repository to the form and it is automatically visualized as an entry field with a label. All rules and triggers are applied as usual. Because the entry field is a component, it has properties such as height and width and associated events.

You can also modify the component definition of a field to make it appear as a spin edit box, an entry field with a prompter, a check box, a set of radio buttons, a list, a track bar or a a progress bar.

The automatic conversion of fields is a major benefit because you do not need to resize and redefine countless attributes as you do when moving from the IBM i to many other visual tools.

The component characteristics of a field are ignored when you use them in your LANSA for i or universal UI applications.

# **Reusable parts**

Component technology is designed to provide productivity, quality and consistency gains by centering the development effort on the creation of standard, reusable, automated building blocks, called reusable parts, from which applications are assembled. As much of the maintenance is carried out at the reusable part level, the time required in testing and verifying individual applications is also greatly reduced.

A typical Visual LANSA application is built from a number of reusable parts defined in the repository, such as lists, fields and standard dialogs. You can change an individual reusable part and this change is reflected in every application that uses the component—the applications themselves do not need to be changed or recompiled. For example you could change the label of a

reusable part and this change would be reflected immediately in every application where the button is used.

Reusable parts can contain logic. You could, for example, create an automated reusable part called EmpList, define it as a list box and write the code to fill it from a specific database table.

Employee	Employee		SELECT	FIELDS (#EMPNAME) FROM_FILE (EMPLOYEE			
Black	Gillian		ENDSELECT	TO_LIST(#EmpList)			
Blake	John		ENDROUTINE				
Bloggs	Fred						
Brown	Freddy						
Brown	Veronica						
Dickens	Charles						
Douglas	Adam	-					

After creating this component, you would reuse it in every application that needs this list of employees. Because the component uses its own code to fetch the entries from the file and display them, no coding is required to fill the list in the application itself.

Later on you could change the part to also display for instance an employee number in the list. Or if the employee information was moved to another file, you could simply change the file name in the SELECT statement of the component. You would then recompile and test the component and store it back in the repository. Again, all these changes would immediately be reflected in every application in which the component is used, without having to recompile and test them individually.

By using reusable parts, you can be sure that your applications look the same and behave the same way. For example by using a standard set of OK and Cancel buttons, you know that your buttons on every screen are the same size, shape and color, have the same caption in the same font and behave consistently.

You can create your reusable parts from scratch or you can copy and modify existing ones. Once you have a standard set of components for your site, creating new applications and extending existing applications becomes a matter of simply assembling together the required parts.

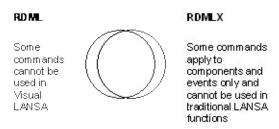
# 6.5 RDMLX Is Extended RDML

Visual LANSA introduces a number of commands to support component definition and event-driven programming. These commands are called RDMLX commands (the X stands for extended). You cannot use them in traditional LANSA functions.

Similarly, there are some existing RDML commands which you cannot use in Visual LANSA. These are mainly commands that deal with the display of data (DISPLAY, REQUEST, POP\_UP) or modes (SET\_MODE, IF\_MODE).

For more information, refer to 6.9 Event-Driven Programs.

Even though there are some differences in how you create RDML and RDMLX programs, the vast majority of RDML commands are used in Visual LANSA. This provides you an easy transition to event-driven programming and ensures that your investment in LANSA skills is maintained.



RDMLX is a 4GL and vastly simpler than other object-oriented languages. It is designed to provide the benefits of graphical user interfaces and event-driven programming for business-focused application development.

### 6.6 Move Existing Applications to Visual LANSA

In most cases you will only convert a small part of your existing application to Visual LANSA components. The bulk of the application probably should continue to use the universal user interface of LANSA, ensuring execution on green screens, Web browsers and Windows.

Identify the forms your users really want to exploit Windows functionality. There might not be that many. Then use Visual LANSA to rapidly add explorerstyle menus, graphs, and grids to them.

# 6.7 A Few Things You Need to Know

This section outlines some basic information about LANSA functions and components.

## Visual LANSA Is Now Used to Create Two Kinds of Applications

You can now create both LANSA functions (Universal UI) and Windowsoptimized Visual LANSA applications.

LANSA functions and components share:

- Fields
- Files
- Most RDML commands
- The editor (a graphical development environment) in which you edit functions, forms and other components.

## You Create Functions and Processes Exactly as Before

There is no change to how you create processes and functions:

• You use the editor to create functions just like before. No new commands, nothing has changed.

## **Component Applications are Created as Forms**

Applications are created as forms (in the same way as Universal UI applications are created as functions). When you want to create a new application, you create a form. When you want to change the application, you edit the form. There are no processes in Visual LANSA.

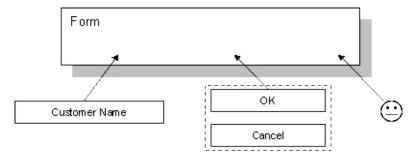
### **6.8 Component Basics**

6.8.1 Designing Visual LANSA Interfaces

- 6.8.2 Components
- 6.8.3 Properties
- 6.8.4 Events
- 6.8.5 Methods
- 6.8.6 Inheritance
- 6.8.7 Components and Instances
- 1 6. Creating Applications Using Components

## **6.8.1 Designing Visual LANSA Interfaces**

You create the interface of your application by dragging components from the repository to forms. A form is itself a component which corresponds to a window of your application when it is running. You can also put various purely visual things such as icons, bitmaps and videos on forms.



↑ 6.8 Component Basics

## 6.8.2 Components

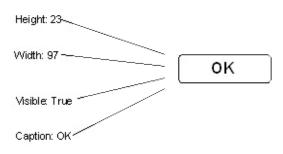
Visual LANSA has the following types of components:

- Fields. Any LANSA field you use in an application becomes a component.
- Forms. A form corresponds to a window of an application.
- Controls. These are the graphical widgets put on the form. Examples of controls are command buttons, list boxes, videos, labels, tree views, radio buttons and track bars.
- Reusable parts. These parts contain controls and code which can be reused in forms (or other reusable parts).
- Visual Styles. This is a special kind of component which controls the appearance of individual fields, forms and controls or entire applications.
- Bitmaps and icons. These are images which are shared by many applications.
- ActiveX controls. These are third-party standard reusable components.
- WAMs. These components use XML to deliver technology services for a variety of client computing devices, including browsers.

1 6.8 Component Basics

### **6.8.3** Properties

All Visual LANSA components have properties which define their characteristics. Most properties deal with the way a component is displayed on the screen: its size, color, and whether it is visible or not. You can set the properties when you are designing your application, or programmatically when the application is running. Often you do not need to change any of the default property values.



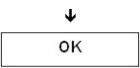
↑ 6.8 Component Basics

## 6.8.4 Events

Traditional LANSA functions (Universal UI) implement procedural programming method, Visual LANSA applications are event-driven.

The main difference in writing Visual LANSA applications compared to traditional LANSA functions is that in Visual LANSA the code belongs to the various controls on the form instead of a single function. This code is run when the user performs an action on the control, for example clicks a button. This is called event-driven programming.

An event is a thing that happens or takes place. Typically an event is any action the user takes. The most common event is click (when the user clicks on a component with the mouse). You could for example define a button 'Print' and write code for the button's click event to print something. When the application is running, every time your user clicks on the Print button the print routine will be executed.



Other common events are:

- Initialize. This event is triggered when a component is first displayed.
- GotFocus and LostFocus. This event is triggered when the user selects a component with the mouse.
- Change. This event happens when the user changes the contents of component, for example a field.

For more information, refer to 6.9 Event-Driven Programs.

## 6.8.5 Methods

A method is how you tell a component to do something. For example to display a form, you use the ShowForm method:

INVOKE #FormB.ShowForm

You can also define custom methods to make a component perform an action.

## 6.8.6 Inheritance

Forms and reusable parts have a property called Ancestor. It is used to select the component from which this component inherits from.

By default all forms inherit from #PRIM\_FORM which is the elementary Visual LANSA form. #PRIM\_FORM cannot be modified. However, you can specify any other form as the ancestor of a form.

The inheriting form will look exactly like its ancestor form: it inherits all the components of its ancestor form. If you make changes to the ancestor form, these changes are reflected in the inheriting form.

You can change the properties of the inherited form and its components. When you change the property of an inherited component, it permanently overrides the original value. So for example if the form has an ancestor form which contains a push-button with the caption 'Save' and you change this to 'Save Changes' on the inheriting form, this caption will not change when you change the caption of the button on the ancestor form.

By default reusable parts inherit from #PRIM\_PANL, a panel on which you can put other components (or reusable parts). You can make any Visual LANSA control, such as a button, tab folder, tree view etc. the ancestor of a reusable part. You can also use one reusable part as the ancestor of another reusable part.

## **6.8.7** Components and Instances

In the object-oriented model of Visual LANSA there is an important distinction between a component and an instance of a component.

When you include a component in a form, you do not include the actual component, only an instance of it. The component serves as a model for the instance: all instances of a component have the same properties, events and methods as the component. However, you can change property values and create event routines for a specific instance. When you do so the component itself remains unchanged.

Let's use the push-button component as an example. Every time you drag a button to a form, it will have the same:

- Properties such as Height, Width, Visible. Initially some these properties have default values, others depend on where the button is placed.
- Click and Initialize events. Initially there is no code for the events.

You then modify some of the properties, usually at least the caption (for example OK), and you write logic in the Click event of the button.

So every instance of a button in your application shares the characteristics of a button component but has its own individual character as well.

The same happens when you drag fields to a form. However, most often you use the instance of a field as such without changing it in any way.

Similarly, instances of a form can be included in another form. For example you might create a standard form used by many applications. To use this form in an application you include it in the application's main form (owner form). The included form is an instance of the standard form. You can modify the properties of the instance, for example its height and width or its caption, but the standard form does not change.

# You Can Have Multiple Instances of a Component on the Same Form

In the case of controls, you often have several instances of a component on the same form, for example push-buttons, check boxes or tab sheets. The instances have different names. For example two push-buttons on the same form are named #PHBN\_1 and #PHBN\_2 by default.

You can also include multiple instances of a field on the same form, but there is seldom reason to do so.

You include several instances of a form in an owner form when you want to be able to display several identical windows simultaneously. For example you might want your application to have more than one window showing employee details open at the same time. In this case you include several instances of the form on the owner form.

## **Changing the Component**

You can change the component definition of a field or a form. You cannot change the definitions of controls (for example buttons, lists, tab sheets). They are predefined.

When you change the definition of a field or a form, their instances also change, except for cases where the instance specifically overrides the change.

For example when you change the description of the field in the field definition, the descriptions of all the instances of the field on all forms are changed. So the instance is not just a copy, it is actively linked to the component.

The description would remain unchanged only if in your application you had specified to use a caption which was specific to the instance.

Similarly, when you change the definition of a form, for example its caption, the caption of all its instances is changed. Again, this does not happen if an instance has an overriding caption.

## **6.9 Event-Driven Programs**

Visual LANSA programs are event-driven whereas traditional LANSA (Universal UI) functions are procedural. This section describes the basic differences between these two programming methods. It also introduces the new commands used in Visual LANSA, called RDMLX, and lists the RDML Commands that cannot be used in Visual LANSA.

If you are totally new to Visual LANSA, you may want to skip this section now and return to it after you created and executed a few Visual LANSA forms.

#### Also see

6.9.1 Dynamic and Flexible Programs

6.9.2 What Is Different Between Event-Driven and Procedural Programs?

6.9.3 How Is the Code Different in a Function and a Form?

6.9.4 A LANSA 'Work with' Function and Form

6.9.5 RDML and RDMLX Programs

↑ 6. Creating Applications Using Components

## 6.9.1 Dynamic and Flexible Programs

In a graphical user interface (GUI) you interact with the program using graphical controls such as buttons, lists, or check boxes. To make it possible for these controls to respond to your actions, an event-driven programming method is required. In this method every control can have its own response to an action, called an event. The event is a notification to the program that something has happened, for example that you have pressed a button, selected an item in a list, or ticked a check box.

Because every action of every control, in other words every event, can have its own logic, the program can respond separately to every single user action on every individual control. This makes it possible for event-driven programs to respond to actions dynamically.

Only the individual piece of logic associated with the event is performed when you trigger an event. This means the events can be executed in any order. This makes event-driven programs very flexible.

Many of the controls used in a GUI have powerful built-in functionality. They provide effective ways of interacting with the user and presenting and navigating data. As a result, you can create extremely compact applications with rich visual communication. Applications like this are easy to learn and to use.

↑ 6.9 Event-Driven Programs

# **6.9.2 What Is Different Between Event-Driven and Procedural Programs?**

There are some fundamental differences in how event-driven and procedural programs are designed and executed:

• In Visual LANSA you create the graphical user interface using controls which have powerful built-in functionality such as tree views, tabbed folders, video and animation.

In a LANSA function you have very limited graphical controls in order to accommodate non-graphical platforms.

• In Visual LANSA you create the interface of the program graphically by dragging controls to a form which corresponds to a window of the application when it is running.

In a LANSA function you create the interface programmatically using DISPLAY, REQUEST and POP\_UP statements.

• In an event-driven program you write code for the individual controls, for example a button. So your application consists of many event routines. These event routines are executed individually which makes it possible for the individual controls on the form to respond dynamically to user actions.

A LANSA function is one program which runs from the beginning to the end. A screen is always processed as a whole so it cannot change dynamically.

• When executing Visual LANSA applications you perform actions directly on objects.

In Universal UI functions you first choose an action, such as Add, which sets a mode in the program and then work on the object.

#### Also see

The Code Is Always Associated With a Component Modes Are Not Needed because Actions Are Performed Directly The Code for a Mode Often Becomes an Event Routine Sequence of Actions Is Free You Create the Interface Graphically

## ↑ 6.9 Event-Driven Programs

## The Code Is Always Associated With a Component

A Visual LANSA application consists of several individual event-routines which are associated with specific components.

This is quite different from a LANSA function which is a single program in which DISPLAY, REQUEST and POP\_UP commands create entire screens. These screens are always processed as a whole.

In contrast, in Visual LANSA every component on a form can have its own event routine which gets executed independently when the user performs a certain action. Because of this, you can create dynamic interfaces which instantly respond to user actions. For example you can disable and enable controls depending on the user's choices:

-

In a LANSA function you cannot change the screen dynamically because it is processed as a whole. The information entered on a screen is processed and validated only when the user clicks on the OK button:

Maintain Marria	ge Status Information			
ID number Marriage Status Spouse Name Marriage Date Divorce Date	A0002 D VERONICA BROWN 1/02/99 0/00/00			
ОК	Prompt Msgs	Exit	Cancel	Help

## **Modes Are Not Needed because Actions Are Performed Directly**

In event-driven programs information is handled directly. For instance if you want to check employee details, you display a list of employees, select the person and view the details. If you want to make any changes, you can do them there and then. The changes are written to the database when you click a button which has an event routine containing an UPDATE statement. Modes are not needed.

In an RDML function you can display the employee details in the same way, but if you want to make any changes, you have to first click a Change button or enter an option number to bring up a screen with input capable fields and to set the mode to \*CHANGE in the program. In other words you signal what action you are about to perform before you actually perform it.

Modes in RDML control whether user input is allowed. In Visual LANSA this is handled using the ReadOnly property (True/False) of the component.

#### The Code for a Mode Often Becomes an Event Routine

What you would write an IF\_MODE statement in a LANSA function often corresponds to what you write in an event routine in Visual LANSA. For instance, in a LANSA function you would normally delete an item by writing a DELETE statement in the IF\_MODE statement testing for the \*DELETE mode. In Visual LANSA you would put the DELETE statement in the Click event of the Delete button:

#### LANSA function

#### Visual LANSA form

IF\_MODE IS(\*DELETE) DELETE FROM\_FILE(PSLMST) ENDIF EVTROUTINE HANDLING(#DEI DELETE FROM\_FILE(PSLMST ENDROUTINE

Just as the code in the in the IF statement only gets executed when the condition is true, the code in the Click event of the Delete button only gets executed when you click the button.

You will probably find that your programs become simpler when they are event driven because there will be fewer nested control structures as the code is broken down to event routines. Also, it is easy to follow a program when you can see straight away which component and event will cause which action.

In the simplest case, when you convert an existing LANSA function to a Visual LANSA form you can:

- Create buttons that correspond to the modes used in the function, for example Add, Change and Delete buttons.
- Copy the RDML from the IF\_MODE statements to the Click events of the corresponding buttons.

## **Sequence of Actions Is Free**

When you perform an action on a component in Visual LANSA, the program executes the individual event routine associated with that action for the component. When the event routine has completed, the program waits for you to decide which event to perform next. The event routines are independent and need not be executed in any predefined order. This makes it possible for you to freely decide what actions to take and in which order.

A LANSA function is less flexible. This is partly because the program executes from beginning to the end in a predefined order. What input you can provide and what options are available depend on what has been specified in the program at that point in the execution. In addition, accomplishing a task usually requires several steps because only one type of action can be performed per screen (for example select, add, change or delete). The order of these screens is predefined.

## You Create the Interface Graphically

When you create a Visual LANSA application, your focus is on the interface. You can choose from many controls with different functionality, and you have a lot of freedom in how you arrange and organize information.

Creating the interface is easy because you do it by dragging and dropping fields and controls to a form with the mouse. The form corresponds to a window in the application when it is running, so you can see instantly what your application will look like. Because creating a window is very quick, you probably try out several alternatives.

When you create an interface for a LANSA function, you do it programmatically using commands such as REQUEST and DISPLAY. You do, of course, use the screen painter to check the interface, but your choices are limited and they concern more the layout than the functionality.

To create effective graphical interfaces, you need to know the functionality and purpose of the different controls. See the descriptions of the individual controls.

To see an example of a graphical user interface, execute the VL\_DEM20 demo application. (To do this, go to the Fields/Components list, locate and select VL\_DEM20 and then choose the Execute command from the ribbon.)

## 6.9.3 How Is the Code Different in a Function and a Form?

Let's return to the marriage status example to see the differences between the code used in a LANSA function and a Visual LANSA form. Both the example function and the example form can be used to change and delete marriage status information.

Note that all validation rules for both the function and the form are defined as rules in the repository.

## **LANSA Function**

You can use the function to display marriage status information for people identified by an identification number (#IDNO). You enter the number on a selection screen after which you can view and delete information. When you click the Change button a screen with input capable fields is displayed:

Maintain Marria	je Status li	nformation			
ID number Marriage Status Spouse Name Marriage Date Divorce Date	A0002 D VERONIC 1/02/99 0/00/00	- 81			
OK If marriage status i	Prompt s D, the date	Msgs of divorce mu	Exit st be entered.	Cancel	Help

Note that this screen does not indicate in any way which fields are mandatory for a particular marital status. Validation rules are applied only when you click the OK button.

## Source for the Function

Here is the source for the function:

```
FUNCTION OPTIONS(*NOMESSAGES *DEFERWRITE *DIRECT)
GROUP_BY NAME(#PANELDATA) FIELDS(#IDNO #STATUS #SPOUSE ;
******** COMMENT(Loop until user EXITs or CANCELs)
BEGIN_LOOP
R10: REQUEST FIELDS(#IDNO) DESIGN(*DOWN) IDENTIFY(*DESC)
******** COMMENT(Fetch file MARRIAGE details )
FETCH FIELDS(#PANELDATA) FROM_FILE(MSTATUS) WITH_KEY(#II
******* COMMENT(Display results to the user )
```

```
SET_MODE TO(*DISPLAY)
POP_UP FIELDS(#PANELDATA) DESIGN(*DOWN) AT_LOC(*UPPER 7)
IF_MODE IS(*CHANGE)
UPDATE FIELDS(#PANELDATA) IN_FILE(MSTATUS)
ENDIF
IF_MODE IS(*DELETE)
DELETE FROM_FILE(MSTATUS) ISSUE_MSG(*YES)
ENDIF
END_LOOP
```

## Visual LANSA Form

The Visual LANSA form has exactly the same functionality as the previous application. In addition it indicates which fields are mandatory by enabling and disabling them based on marital status. This instant feedback guides the user to enter the required information:

Employee Number	A001	Fetch		
Married	Spouse name:	Veronica Brown		
Never Married	Marrige date:	14/11/2011	-	
Divorced	Divorce date:		-	
		-		Save
				Delete

#### Source for the Form

Here is the source for the form:

\* Component Definitions

DEFINE\_COM CLASS(#IDNO.Visual) NAME(#IDNO) DISPLAYPOSITION DEFINE\_COM CLASS(#SPOUSE.Visual) NAME(#SPOUSE) DISPLAYPOS DEFINE\_COM CLASS(#MARRIED.Visual) NAME(#MARRIED) DISPLAY DEFINE\_COM CLASS(#DIVORCED.Visual) NAME(#DIVORCED) DISPL DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#SAVE) CAPTION('Save') I DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#DELETE) CAPTION('Dele DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#FETCH) CAPTION('Fetch' DEFINE\_COM CLASS(#PRIM\_GPBX) NAME(#FETCH) CAPTION('Mari DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#RDBN\_1) CAPTION('Mari DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#RDBN\_2) CAPTION('Mari DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#RDBN\_3) CAPTION('Mari DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#RDBN\_3) CAPTION('Div( DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#STBR\_1) DISPLAYPOSITI

\* Other definitions GROUP\_BY NAME(#PANELDATA) FIELDS(#STATUS #SPOUSE #MARR EVTROUTINE handling(#com\_owner.Initialize) SET #com\_owner caption(\*component\_desc) ENDROUTINE

```
* Event routines
* Fetch file mstatus details
EVTROUTINE HANDLING(#FETCH.Click)
FETCH FIELDS(#PANELDATA) FROM_FILE(mstatus) WITH_KEY(#IDNC
if cond('#status *eq N')
set com(#RDBN_1) ButtonChecked(True)
set com(#spouse #married #divorced) enabled(false)
endif
if cond('#status *eq M')
set com(#RDBN 2) ButtonChecked(True)
set com(#spouse #married) enabled(true)
set com(#divorced) enabled(false)
endif
if cond('#status *eq D')
set com(#RDBN 3) ButtonChecked(True)
set com(#spouse #married #divorced) enabled(true)
endif
ENDROUTINE
```

\* Enable and disable fields according to marital status

EVTROUTINE HANDLING(#RDBN\_1.Click ) set com(#spouse #married #divorced) enabled(false) set com(#spouse) value(\*blank) set com(#married #divorced) value(\*zero) change #status 'N' ENDROUTINE

EVTROUTINE HANDLING( #RDBN\_2.Click ) set com(#spouse #married) enabled(true) set com(#divorced) value(\*zero) set com(#divorced) enabled(false) change #status 'M' ENDROUTINE

EVTROUTINE HANDLING(#RDBN\_3.Click) set com(#spouse #married #divorced) enabled(true) change #status 'D' ENDROUTINE

\*Save details EVTROUTINE HANDLING(#Save.Click) UPDATE FIELDS(#PANELDATA) IN\_FILE(mstatus) ENDROUTINE

EVTROUTINE HANDLING(#DELETE.Click) DELETE FROM\_FILE(mstatus) ISSUE\_MSG(\*YES) ENDROUTINE

END\_COM

Code Comparison ↑ 6.9 Event-Driven Programs

## **Code Comparison**

The following tables outline the similarities and differences in the two applications.

1. Both the function and the form start with a FUNCTION statement:

LANSA Function	Visual LANSA form
N/A	A BEGIN_COM statement (with a matching END_COM statement in the very end) and several lines of DEFINE_COM statements
	These statements define the form and the controls on it. They are inserted automatically when the interface is created. Normally you can ignore all the definition statements.

#### FUNCTION OPTIONS(\*DIRECT)

2. Both the function and the form have a GROUP\_BY statement:

LANSA Function	Visual LANSA form
A BEGIN_LOOP statement with a matching END_LOOP statement at the very end.	N/A. There is no loop structure. Event routines can be executed in any order.
REQUEST statement	N/A. The interface has been created graphically at design time.

3. Both the function and the form have a FETCH statement to fetch details for the specified person. In the form the statement is in the Click event of the Fetch button.

LANSA Function	Visual LANSA form
SET_MODE TO(*DISPLAY)	N/A. Modes are not used.

statement.	
POP_UP statement	N/A. The interface has been created graphically at design time.
N/A. Screens cannot be modified dynamically.	Event routine executed every time a radio button is clicked which enables and disables fields according to marital status
IF_MODE IS(*CHANGE) structure performing database updates	Event routine executed when the Save button is clicked. It contains the same UPDATE statement as the corresponding IF_MODE IS(*CHANGE) structure.
IF_MODE IS(*DELETE) statement which deletes the record	Event routine executed when the Delete button is clicked. It contains the same DELETE statement as the corresponding IF_MODE IS(*DELETE) statement.

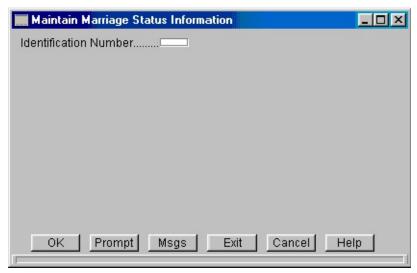
↑ 6.9 Event-Driven Programs

## 6.9.4 A LANSA 'Work with' Function and Form

The previous example was simplified to make a code comparison easy. To give you a more realistic view of what can be done with the two programming methods, here is an example of a 'Work with' type of application implemented as a LANSA function and as a Visual LANSA form.

Both applications are used for maintaining information in the sample PSLMST file. They use a logical view of the file which is keyed by employee surname. You can use them to view, edit and delete employee details.

The LANSA function was created using the FRMWRK1 template. It has a screen where you can limit the employees to be shown by entering a partial surname:



When you click OK, a Work with Employees screen is displayed. This screen shows a list of employees whose names match the partial name. You can view, modify or delete employee information by entering a number in the Opt field.

Work with Employees	
Type options (and Employee), press Enter.	
2=Change 4=Delete 8=Details	
Employ Opt NumberSumame A0111 SMITH A1003 SMITHE A1006 SMITHERS A1005 SMITHS A0193 SMITHSON A1004 SMITHSON A1004 SMITHSON A1002 SMYTHE A1009 SNASHALL A1009 SNASHALL A1008 SNEDDON A1007 SNELL	
OK Prompt Msgs Forward Bkwd Cancel Help	

When you choose the Change option, a screen with input capable fields is displayed.

Work with Employee Details	<u> </u>
Employee NumberEmployee SurnameEmployee Given Name(s)Employee Given Name(s)	
OK Prompt Msgs Cancel Help	

Similarly, selecting the Details and the Delete options brings up a screen showing the same information but with input protected fields.

Note that:

- This application consists of several screens. One screen can only be used for one specific purpose (often corresponding to the mode: change, delete or display).
- The sequence in which the screens are displayed is predefined. For example when you have changed employee details, the Request Employee to Work with Screen is displayed. When you have viewed the details, the Work with Employee screen is displayed.

Depending on what you are doing this might or might not be the order in which you would want the screens to be displayed but you cannot change it.

• Because of the number of screens, there is quite a bit of navigation involved. For example comparing employee information is difficult because you must bring up a separate window to view the details individual employees.

The Visual LANSA application consists of a single form. It performs the same actions as the LANSA function above. It has:

- A list of employees
- Several fields
- A button to save changed employee details, a button to add a new employee and a button to delete employee records.

Employee Black Blake Bloggs Brown Dickens Douglas Jackson Jones Lewis Lincoln Maxwell McCully Moore Morrison Neave Pattinson Paul Perron	Employee Giv Gillian John Fred Freddy Veronica Charles Adam Diane Shirley Tony Paul Andrew Lisa John Alan Gary George Patrick Christophe	<ul> <li>k. Employee Number</li> <li>Employee Surname</li> <li>Employee Given Name(s)</li> <li>Street No and Name</li> <li>Suburb or Town</li> <li>State and Country</li> <li>Post / Zip Code</li> <li>Home Phone Number</li> <li>Business Phone Number</li> <li>Start date (YYMMDD)</li> </ul>	A1018 Blake John 45 Colby Place, Neutral Bay NSW 2021 121 1300 231 1919 85/06/01 Save Delete
--	---	---	--

Note that:

- This application consists of a single window, so all information is visible immediately.
- No navigation is required. You can use one and the same window to filter employees by surname, select an employee, display details, change the details and delete the details.
- You can perform actions in whatever order you want. For example it is up to you to decide when (and if) you want to change the partial surname.
- You can check and compare information quickly and easily.

An application like this can only be created because individual controls have their own event routines. Because the event routines are executed individually, the window can change dynamically according to user actions:

• When you press the Get button, a SELECT statement is run and the list is filled:

EVTROUTINE HANDLING(#Get.Click) change #surname #partname clr\_list #list\_1 SELECT FIELDS(#list\_1) FROM\_FILE(PSLMST2) WITH\_KEY(#SURNAME) GENERIC(\*YES) add\_entry #list\_1 ENDSELECT ENDROUTINE

• When you select an employee in the list, the details for that employee are fetched:

EVTROUTINE HANDLING(#List\_1.ItemGotSelection) FETCH FIELDS(#ALLFLDS) FROM\_FILE(PSLMST) WITH\_KEY(#EM ENDROUTINE

• When you click the Save button, the details are updated:

EVTROUTINE HANDLING(#SAVE.CLICK) UPDATE FIELDS(#ALLFLDS) IN\_FILE(PSLMST) WITH\_KEY(#EMPN ENDROUTINE

• When you click the Delete button, the details are deleted:

EVTROUTINE HANDLING(#DELETE.CLICK) DELETE FROM\_FILE(PSLMST) WITH\_KEY(#EMPNO) ENDROUTINE

An application like this could not be implemented as a LANSA function because in a function the entire screen is processed as a whole.

1 6.9 Event-Driven Programs

## 6.9.5 RDML and RDMLX Programs

#### RDML Commands that cannot be used in Visual LANSA

When you are writing event-driven programs, the majority of the commands you will use are RDML commands. However, there are a number of commands, called RDMLX commands, which are required for the event-driven programming method.

The RDMLX commands are used to invoke methods, and to handle events and properties. These are all high level 4GL commands which shield you from many of the complexities typical of object-oriented programming. Some of these RDMLX commands are:

<b>RDMLX Only</b>	Description
BEGIN_COM and END_COM	Starts and ends a component definition, usually a form. Inserted automatically.
DEFINE_COM	Defines a component inside another one. Inserted automatically.
EVTROUTINE	Defines a routine for an event. Inserted automatically.
INVOKE	Invokes a method.
SET	Sets a property.
DEFINE_EVT	Creates a user-defined event.
SIGNAL	Triggers a user-defined event.
DEFINE_MAP	Creates input and output parameters.
DEFINE_PTY	Creates a user-defined property.
MTHROUTINE	Creates a user-defined method.
PTYROUTINE	Creates a user-defined property.

The BEGIN\_COM, END\_COM, DEFINE\_COM and EVTROUTINE are automatically inserted by the Visual LANSA editor when you drag and drop components or click on events, so you will never have to enter them yourself. SET is one of the few RDMLX commands you will be using frequently:

SET COM(#MyEditBox) TEXT('HELLO!')

This command sets the Text property of MyEditBox to 'HELLO!'

#### Also See

6. Creating Applications Using Components

↑ 6.9 Event-Driven Programs

**RDMLX** Commands in the *Technical Reference Guide*, for a list of all RDMLX commands.

## **RDML** Commands that cannot be used in Visual LANSA

Because of the difference in nature between the existing procedural LANSA programs and the event-driven programs there are some RDML commands that cannot be used in Visual LANSA:

#### **RDML Only Description**

DISPLAY, REQUEST, POP_UP	In Visual LANSA you design the interface graphically.	
SET_MODE, IF_MODE	Modes are not used because actions can be accomplished directly.	
IF_KEY	These involve DISPLAY / REQUEST / POP_UP activities.	
MENU	Procedural transfer of control activities.	
TRANSFER	Procedural transfer of control activities. This command can be used with WAMs.	
DEF_LIST	For browse lists. Working lists may be defined. In Visual LANSA you use list-type components such as list views, grids, and tree views instead.	
All Reporting	Reports cannot be produced in RDMLX. Code reporting logic in RDML functions and call them from RDMLX components.	
RENAME	Not supported because its use may conflict with class usage rename activities.	

↑ 6.9.5 RDML and RDMLX Programs

## 6.10 Work with Components

- 6.10.1 Add and Move Components
- 6.10.2 Accomplish Common Tasks Using Pop-up Menus
- 6.10.3 Copy a Component
- 6.10.4 Modify Default Component Behavior

#### Also See

Using the Editor in the User Guide

Editing Components and Functions in the User Guide.

1 6. Creating Applications Using Components

## 6.10.1 Add and Move Components

You can add components from the Repository tab to a container (form, reusable part, panel etc.) either by drag-and-drop or by double-clicking. The Design view has to be displayed when you do this.

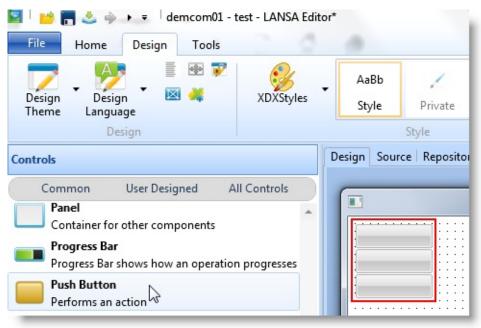
## **Drag-and-Drop**

When you use drag-and-drop you select the component in the Repository tab, keep the left mouse button pressed and position the cursor where you want to place the component and then release the mouse button.

## **Double-Click to Do it Quickly**

Double-click a component in the Repository tab to place it in the top left corner of the container that has focus in the Design view. When you double-click the next component, it is placed underneath the previous component and aligned with it.

Double-clicking is the quickest way to add several components and automatically align them.



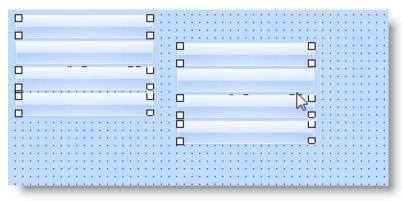
## Select a Group

To select and move a group of components:

- Place the cursor in the top left area corner of the area you want to copy
- Press the left mouse button down and then move the cursor to the bottom

right corner of the area you want to copy.

• Move the components to where you want them placed and release the left mouse button.



1 6.10 Work with Components

## 6.10.2 Accomplish Common Tasks Using Pop-up Menus

You can access common actions for a component by right-clicking it to bring up a pop-up menu. You can do this in the Outline tab or the Design view.

	Primitive: PHBN_1	+
	Delete Component	
	Copy Component	N
	Cut Component	5
	Add <u>P</u> opup Menu	
	Goto Definition	
	Save Definition	+
4	Events : Push button	×

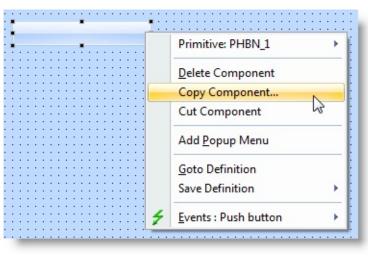
In this pop-up menu you can copy, cut or delete the component (for the Add button). You can also create a pop-up menu for the button or move to the DEFINE\_COM statement used to create the button. The pop-up menu also offers a shortcut to the click and initialize events of the button. The events available in the popup menu vary according to the component.

1 6.10 Work with Components

## 6.10.3 Copy a Component

If you want to copy a form, select its source code and copy it using the Copy command on the ribbon or the context-menu. Then create a new form or visual style and paste the code in it. Copying the source is also handy if you want to store the component or send a copy of the component as a text document.

If you want to copy a component on a form, right-click it in the Design view to bring up a pop-up menu:



A dialog is displayed where you can specify how you want the component copied. If the component is such that it contains other components you can specify whether you want them and their event routines copied as well.

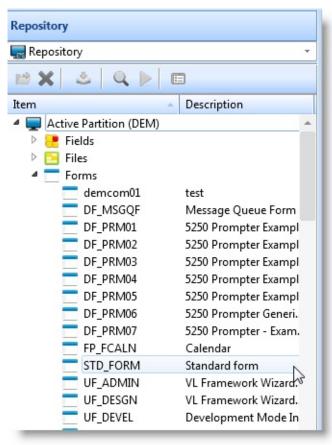
Include		
Component		
Include Event Routines		
Children		
Include Components		
Include Event Routines		
<u>O</u> K <u>C</u> ancel		

Then move the focus to the component you want to paste the copied component and paste it using either the ribbon or Ctrl + V.

## ↑ 6.10 Work with Components

## 6.10.4 Modify Default Component Behavior

Whenever you create a new form, it is modeled on the standard form (STD\_FORM) component in your system. Similarly, all reusable parts are modeled on STD\_PANL.



Often, it is desirable to change default behavior for all objects with the aim of enforcing site standards. STD\_FORM and STD\_PANL are components, just as any forms or reusable parts you create are components, and you could modify them if desired. However, STD\_FORM and STD\_PANL are shipped as part of a default partition definition with LANSA. When a partition is initalized, any changes you have made will be lost.

Rather than modifying the defaults, the best technique is to create your own base form and panel and to use them as the ancestor of all of the objects you create. Create a new form, and change its behavior as required. For example, you may want all forms to appear in the center of the screen by default. To do this, set the Formposition property to ScreenCenter and save the form.

When you create new forms, set the ancestor property to your default form, and

the properties, events and methods that you have defined in your base form will be inherited. Inheritance is a very powerful feature of Visual LANSA. For more information, refer to Ancestor Components.

 $\Uparrow$  6.10 Work with Components

# 6.11 Create an Application

# 6.11.5 Create Applications Using a Wizard

# 6.11.6 Create Applications Using Templates

This section shows you how to create an application used to maintain fields in a file. The application consists of a list of employees and fields showing employee details. You can limit the entries in the list by entering a partial surname. You can use the application to view and maintain the details of existing employees, to add new employees and to delete employee information.

nter partial	surname:	b	Get		
Employe Black Blake Bloggs Brown Brown	Employee G Gillian John Fred Freddy Veronica	Emp A14 A10 A00 A35 A00	Employee Number Employee Surname Employee Given Name(s) Street No and Name Suburb or Town State and Country Post / Zip Code Home Phone Number Business Phone Number Start date (YYMMDD) Termination Date (YYMMD Department Code Section Code Employee Salary	A0090 Bloggs Fred 70 King Street Newtown NSW 2220 344-2234454545 654 6475 X432 92/08/03 D) 0/00/00 FLT 03 20,045.91 \$	Save New Delete

The application uses data from the Personnel System Demonstration PSLMST file which is part of the LANSA DEM partition.

If you want to create this application yourself, you need to create field PARTNAME which uses SURNAME as the reference field. The application is created in these steps:

6.11.1 Step 1. Create the Form

- 6.11.2 Step 2. Create the Interface
- 6.11.3 Step 3. Write the Code
- 6.11.4 Step 4. Compile and run the Program

# 

# 6.11.1 Step 1. Create the Form

You start creating an application by creating a form. The form will become your application's window when it is running.

To create a form select Form from the New option in the File menu or the Repository tab.

The New Form dialog is displayed. Enter the name and the description for the form:

- Name: DEMCOM01. You can create ten components named DEMCOM01 to DEMCOM10 even if you do not have Visual LANSA hardware key.
- Description: Sample Application

In this dialog you can also specify the framework and group where you want to store the form. In this example the form is stored in the Personnel & Payroll framework and in the Demonstration Material group.

New Form		×
Name	DEMCOM01	Create
Description	Sample Application	Cancel
Framework	Administration (ADMINISTRATION)	
Group	·	
Identifier	DEMCOM01	
Enabled For RDMLX		

To start working on the form, click the OK button.

# View the Source Code

If you now bring up the Source view, you will notice that the component definition statement for the form has been created automatically.

#### END\_COM

•••

A Visual LANSA application always starts with a FUNCTION statement followed by a BEGIN\_COM statement defining the form.

The END\_COM statement ends the component definition. It is always the last statement in a Visual LANSA application.

Note that if the standard form (STD\_FORM) definition on your system has been modified, you may see other statements as well. For more information, refer to 6.10.4 Modify Default Component Behavior.

↑ 6.11 Create an Application

# 6.11.2 Step 2. Create the Interface

In Visual LANSA you create the interface of the application before you write the code.

# 2a. Set the Form's Properties

Details		
DEMCOM01		
<b>• • ?</b>		
Properties Events N	/lethods	_
BorderIcons	Maximize+Minimize+Syste	*
Caption		
ClientHeight	302	
ClientWidth	484	
Cursor	*NULL	
🖻 DragStyle	None	
🖻 EnableChildren	False	
nabled 🖉	True	
🖻 EnsureVisible	Application	H
FormPosition	Designed	
🖻 FormStyle	Normal	
🖻 FrameStyle	Sizable	
🦻 Glass	*NULL	
🖻 Height	340	
🖻 Hint		
🖻 HintPopup	*NULL	-
HintShow	True	
🖻 HintTitle		
🖻 Icon	*NULL	
🖻 Image	*NULL	
🖻 ImageAlignment	Center	
🖻 LayoutManager	*NULL	
🖻 Left	487	
🖻 MenuBar	*NULL	
🖉 MouseOverStyle	*NULL	
interest 🖉 🎽 🖉	DEMCOM01	
P Owner	#DEMCOM01	Ŧ

To view the form's properties, select it on the desktop or in the Outline tab. Bring the Details tab topmost. It consists of a list of properties and a list of events for of the form.

The list of properties shows the name of the property on the left and its value on the right.

You can change the value of any of the properties except the name of the form. You do this by selecting a property and then specifying its value.

You can either type in values, or if they are predefined, you can select them from a list.

# Write the Caption

The first property you need to change is the form's Caption property to create a title for your window:

1. Click on the Caption property.

2. Type Employee. As you type it, the new caption is displayed in the title bar of the form.

# Size the Form

Next make the form a bit larger than the default size:

- 1. Select it and move the cursor towards its edge.
- 2. When the cursor changes to a double-arrow, drag the edges with the left mouse button pressed down.
- 3. Alternatively, you can change the Height and Width properties of the form.

You can keep resizing the form as required as you add controls and fields on it.

# 2b. Add Fields and Controls

This section shows you how to add the required fields and controls to the form.

# **#Partname field and a button**

First add the #PARTNAME field which uses #SURNAME as the reference field.



To add the PARTNAME field:

- 1. Start by locating it in the Repository.
- 2. Drag the PARTNAME field to the form or double-click it.
- 3. Bring up the Details tab and make its LabelType property Caption.
- 4. Type 'Enter partial surname:' in its Caption property

# **Command Buttons**

In addition to the Get command button used with the partial surname field, you need these command buttons to work with the employee details:

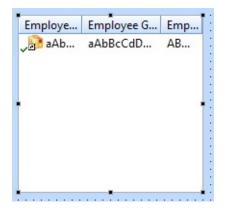


To add buttons to the form:

- 1. Select the Controls tab.
- 2. Drag push button to the form four times. Alternatively you can double-click it four times.
- 3. Change the captions of the buttons.
  - a. Make the caption of the first button Get
  - b. Make the caption of the second button Save
  - c. Make the caption of the third button New
  - d. Make the caption of the fourth button Delete
- 4. Change also the default names (#PHBN\_n) of the buttons to match the caption. In this way it will be easier to identify the buttons when you write event routines for them.
- 5. Position them on the form as shown in the picture in 6.11 Create an Application at the beginning of these steps.

# List View

Next create the list view which shows the employees:



To create the list view,

display again the the Controls tab. Drag a list view control (PRIM\_LTVW) to the form or double-click it. Now you need to create the columns for the list. To do this:

1. Locate the file PSLMST in the Repository tab.

2. Click on the plus sign in front of it to display the fields defined for

Repository	
🔙 Repository	-
📴 New 👻 📂 🗙 🔍	
Item	<ul> <li>Description</li> </ul>
LZ2E	Sales Histor 🔺
EZ2I	Inventory ir
LZ2P	Product
EZ2W	Country Wa
PSLEVENT	Personnel E
PSLIMG	Personnel Ir
PSLMST	Personnel
PSLMST1	Personnel b
PSLMST2	Personnel b
🔑 EMPNO	Employee N
📒 SURNAME 📐	Employee S
🤃 GIVENAME	Employee C
Contraction Addression Addression	Street No ar
💓 ADDRESS2	Suburb or T
ADDRESS3	State and C

this file.

3. Make sure the list view has the focus in the Design view. In the Repository tab double-click these fields to add them to the list view (or drag them):

SURNAME GIVENAME

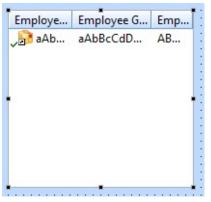
EMPNO

When you add a field to a list, a list column is created.

Display the Outline tab. The list view columns are named LVCL\_n:

- LVCL\_1 is the column created from the #SURNAME field
- LVCL\_2 is the column created from the #GIVENAME field
- LVCL\_3 is the column created from the #EMPNO field

You can rename these fields if you want to use more descriptive names.



In the Design view notice how the list view shows that a column has been added to it.

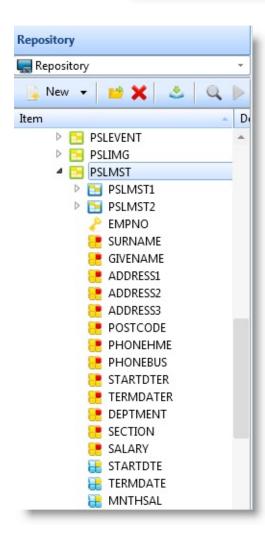
- 1. Now size the columns as shown by dragging them.
- 2. Display the *Details* tab for all three columns and set their *CaptionType* property to *ColumnHeadings*.

This sets the column headings to the value specified in the field definition of the source fields.

# **Employee Details Fields**

You will need these fields showing employee details on the form:

Employee Number	ABCDE
Employee Surname	aAbBcCdDeEfFgGhHiIjJ
Employee Given Name(s)	aAbBcCdDeEfFgGhHiIjJ
Street No and Name	aAbBcCdDeEfFgGhHiIjJkKILm
Suburb or Town	aAbBcCdDeEfFgGhHiIjJkKILm
State and Country	aAbBcCdDeEfFgGhHiIjJkKILm
Post / Zip Code	123456
Home Phone Number	ABCDEFGHIJKLMNO
Business Phone Number	ABCDEFGHIJKLMNO
Start date (YYMMDD)	12/34/56
Termination Date (YYMMDD	)) 12/34/56
Department Code	ABCD
Section Code	AB
Employee Salary	123,456,789.12



To add the fields, again display the PSLMST file in the Repository tab.

Drag and drop these fields to the form:

- EMPNO, GIVENAME, SURNAME (add these fields to the form even though they are in the list to make it simple to add new employees)
- ADDRESS1, ADDRESS2, ADDRESS3 and POSTCODE
- PHONEHME and PHONEBUS
- SALARY
- STARTDTE and TERMDATE
- DEPARTMENT and SECTION

Notice that as you drag the fields to the form, they are displayed with a label specified in the field definition. Also, they are filled with sample data which indicates the required length for the field.

### **Status Bar**

Lastly, you need a status bar for showing messages:

aAbBcCdDeEfFgGhHiljJkKILmMnNoOpPqQrRsStTuUvVwWxXyYzZ

Display the the *Controls* tab and drag a status bar to the form. It will automatically attach itself to the bottom of the form.

# **Finalize the Interface**

Make sure that the interface of the application corresponds to the picture shown in 6.11 Create an Application at the beginning of these steps.

↑ 6.11 Create an Application

# 6.11.3 Step 3. Write the Code

Now that your user interface is complete, it is time to write the code. Click on the Source tab to display the source code for the form. You will notice that the editor has automatically created the DEFINE\_COM statements for the controls and fields you have included on the form. These statements show the properties defined for the components.

# **3a. Create a Group for All the Employee Details Fields**

To make it easier to deal with all the employee details fields, create a group for them:

GROUP\_BY NAME(#ALLFLDS) FIELDS(#EMPNO #SURNAME #GIVEN

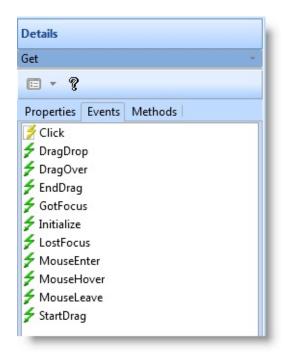
# **3b. Write an Event Routine for the Get Button**

- 1. To write the code to fill the list view according to a partial surname, display the Details tab, select the GET button on the list and then bring the event tab to the front.
- 2. Double-click on the Click event. Notice that editor inserts the EVTROUTINE and ENDROUTINE statements for you in the Source view.

EVTROUTINE HANDLING (#GET. Click)

#### 

The icon in front of the Click event in the Details tab changes to indicate that a routine has been specified for the event:



3. Add the following code in the Click event of the Get button:

```
EVTROUTINE HANDLING(#GET.Click)
change #surname #partname
clr_list #ltvW_1
SELECT FIELDS(#ltvw_1) FROM_FILE(PSLMST2) WITH_KEY(#SURN
add_entry #ltvw_1
endselect
ENDROUTINE
```

The event routine assigns the value entered in the #PARTNAME field to #SURNAME. It then clears the list view. The SELECT loop selects all employees whose surname matches the value of #SURNAME and adds the entries to the list view.

- 4. Write an Event Routine to Fetch the Details for an Employee:
  - a. Create a Click event routine for the ItemGotSelection event of the list view. This event is triggered when the user selects an employee in the list view.
  - b. Now write the following code to fetch the details for the selected employee.

EVTROUTINE HANDLING(#LTVW\_1.ItemGotSelection)

# FETCH FIELDS(#ALLFLDS) FROM\_FILE(PSLMST) WITH\_KEY(#EMP ENDROUTINE

5. Write an Event Routine to Enter a New Employee

Create an event routine for the Click event of the New button. Use the event to blank out all the fields so that the details of a new employee can be entered.

EVTROUTINE HANDLING(#NEW.Click) change #allflds \*Null ENDROUTINE

6. Write the Event Routine to Delete an Employee

Create an event routine for the Click event of the Delete button to delete the currently selected employee:

EVTROUTINE HANDLING(#DELETE.CLICK) DELETE FROM\_FILE(PSLMST) WITH\_KEY(#EMPNO) ISSUE\_MSG(\*` ENDROUTINE

7. Write the Event Routine to Save Changes

Create an event routine for the Click event of the Save button to either update the details of an existing employee or to insert the details of a new employee:

```
EVTROUTINE HANDLING(#SAVE.CLICK)
check_for in_file(pslmst) with_key(#empno) val_error(*next)
if_status is(*EQUALKEY)
UPDATE FIELDS(#ALLFLDS) IN_FILE(PSLMST) WITH_KEY(#EMPN
IF_STATUS IS(*OKAY)
MESSAGE MSGTXT('Employee details have been updated.')
endif
else
INSERT FIELDS(#ALLFLDS) TO_FILE(PSLMST) ISSUE_MSG(*YES)
IF_STATUS IS(*OKAY)
MESSAGE MSGTXT('Employee details have been added.')
endif
endif
```

#### ENDROUTINE

The event routine first checks to see if the employee exists. If it does, it updates the employee details. If no record with the matching employee number is found, it inserts the employee details. A message is displayed after the update and the insert.

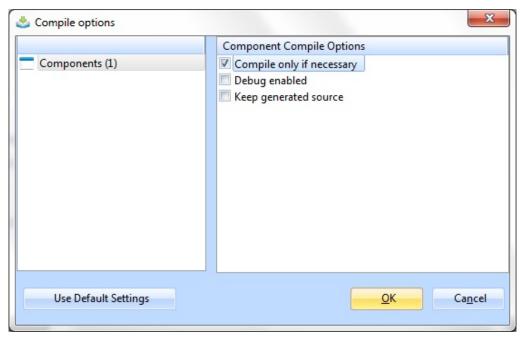
↑ 6.11 Create an Application

# 6.11.4 Step 4. Compile and run the Program

The application is now complete.

1. To compile the program, select the Compile command on the ribbon.

You do not need to keep the generated source or debug enable your component. In the Component Compile Options dialog box specify the options like this:



- 2. Click OK.
- 3. After the compilation has ended, execute your program using the Execute command:



Note that you can also execute the form outside the LANSA development environment using the Exec Form option of the LANSA program folder:

Execute Form on wor	kstation
Form Name	DEMCOM01
Language	ENG
Partition Identifier	DEM
LANSA User	PCXUSER
Database User	USERID
Database Password	
Database Name	LX7EEVAP
Database Type	MSSQLS
Default Printer	LPT1
Debug	N
Debug Host	eevapc:51237
Trace	N
Max Trace Lines	20000
Trace level	4
Trace Categories	ALL
Heap Validation	X
Render Type	x
Graphics Processing	X
OK Cancel	Help Parameter Help

↑ 6.11 Create an Application

# 6.11.5 Create Applications Using a Wizard

You can create LANSA applications modeled on the LANSA Editor using a simple wizard.

The generated application will have:

- An interface capable of handling documents
- A list of proxies (representing documents) in the object browser which the end-user can use to navigate through the list of documents.
- An embedded document editor for viewing and editing documents.
- A File menu with predefined common commands: New, Save, Close, Show Browser, Open Objects...

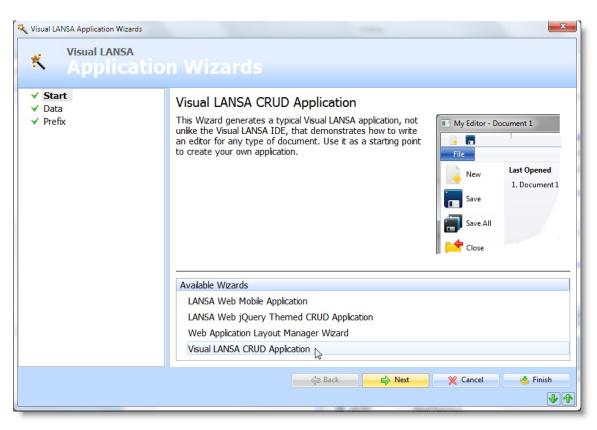
Here is an example of an application created with the wizard:

			My	Editor - Name_3 - Description_3	- 🗆 🗙
File H	lome				۵ (
	Ribbon				
	ews				
Name	Description	····· <u> </u>	warne	Name_3	
Name_1 Name_2	Description Description_1 Description_2		ក្តី Description	Description_3	
Name_3 Name_4	Description_3 Description_4		-		
Name_5	Description_5				
Name_6 Name_7	Description_6 Description_7				
Name_8 Name_9	Description_8 Description_9				
Name_10	Description_10	Proxy			
					Document
	Object Browser				

To run the wizard select *Wizard* in the *Tools* tab of the ribbon:

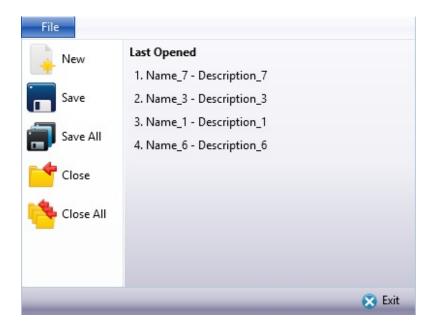
ile Home Design To	ols									
Opploy         Import         Quick Exp	ort VL Framework	LANSA Client	Integrator Studio	Logical Modeler	Message Files	Wizards	L Publish	Translate	Import Translations	Upgrade

And then select the Visual LANSA CRUD Application:



The generated application consists of:

- The main form which is the skeleton of your application. All objects including the ribbon and the menu reside on the main form.
- The ribbon: the *Show Browser* and *Open Objects* commands are placed on ribbon by default. *Show Browser* shows the object browser and *Open objects* shows a list of last opened documents. You can add new commands to the ribbon. For each new command place a push button on the ribbon, assign an icon to it and write a click event for it.
- The menu bar: The menu bar contains menu items for predefined common commands. You can add new menu items and base their code on any of the predefined commands:



- The document: Document is a list of properties to be shown or edited as a whole. If you use the sample data option in the wizard (which is the default option) your document will have the properties of *Name* and *Description*. If you choose to use an existing file, the document will consist of all the fields of that file.
- The proxy: Proxy is a subset of properties of each document which acts as a unique key for accessing that document. What you see in the Object Browser is a list of proxies. Because documents are worked on one at a time, there is no point in loading all documents at once. Only the proxies are loaded and when a proxy is selected, the document associated with it is loaded. In the standard example the proxy and the document have the same number of properties ("Name" and "Description"), but normally a document contains more data.
- Object Browser: Object browser shows a list of all proxies.

You can see a list of the generated objects on the last page of the wizard (prefixed "MY\_" prefix by default; you can change it):

- MY\_DATA: This class defines your data model class with the properties of your document. If you want to change document properties you can make changes here. The generated application uses this class for both proxy and document objects.
- MY\_STORE: This class contains a list of MY\_DATA objects representing all documents. It has functionality to insert, delete and update a document or to find a document using its proxy.
- MY\_EDIT: This is a reusable part providing the visual representation of the

document to edit.

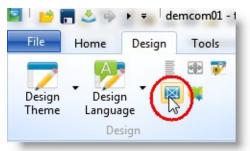
- MY\_BRWSR: This is a class that handles the proxy list as described above using a tree view.
- MY\_MAIN: This is the main form of your application where the ribbon, menus and all other objects reside. You can place new ribbon buttons or add new menu items in the code and start adding new features to your application.

# **6.11.6 Create Applications Using Templates**

The preceding steps described the manual creation of an application. Note that a quicker and easier way of creating a Visual LANSA application is to use a template wizard. There are templates for:

- Maintaining a database file (VL\_BASEMNT). This template creates a simple form which can be used to insert, update, fetch and delete field information. Optionally it creates a tabbed folder.
- Working with a component (VL\_BASEWRK). This template has the same functionality as VL\_BASEMNT plus a tree view for navigating data and a status bar. This template is fairly complex.
- Creating a tabbed folder with four sheets (VL\_BASETAB).
- Creating a tree view and a menu (VL\_BASETRE).
- Creating a standard menu (VL\_BBMNUBR). You can of course modify the menu options after the menu has been created.
- Creating a status bar and two subroutines for displaying status and information messages (VL\_BBSTSBR).

To run a template, use the *Templates* command in the *Design* tab of the ribbon.



## ↑ 6.11 Create an Application

# 6.12 Enrolling Bitmaps and Icons

When you want to use icons or bitmaps in many of the Visual LANSA controls such as tree and list views, forms, push-buttons and toolbar buttons, you must first enroll the image in the repository as a bitmap or icon component.

Using repository-enrolled images makes development and application deployment easier because the images are managed centrally so no separate image files need to be maintained or distributed. Also, you do not need to recompile your forms when you change an image.

There is a set of icons and bitmaps supplied with LANSA (called VB\_xxxxx and VI\_xxxxx).

Visual LANSA does not provide an icon editor, but many cheap or free icon editors are available on the web.

Libraries of icons are also accessible on the web. Usually these libraries are free or available for a nominal charge.

#### Also see

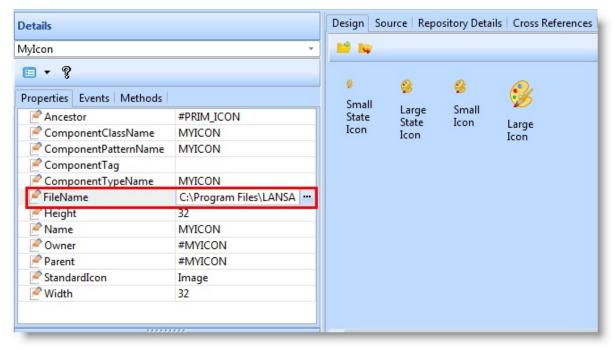
- 6.12.1 Enrolling an Image
- ↑ 6. Creating Applications Using Components

# 6.12.1 Enrolling an Image

To enroll an icon or a bitmap create a component of type icon or bitmap:

New Icon			X
Name	MyIcon		Create
Description	Test application icon		Cancel
Framework	Administration (ADMINISTRATION)	*	cancer
Group		*	
Identifier	MYICON		

The Design view of the editor will show the icon or bitmap and the Details tab can be used to set the image properties. The image file is inserted using the FileName property.



Note that after the picture is inserted, the FileName property is blanked out to indicate that no link is retained to the original image file.

Refer to the Features help for detailed information about the image properties. The description of 6.15.5 List View shows how to specify images in list and tree views.

1 6.12 Enrolling Bitmaps and Icons

### 6.13 Forms

- 6.13.1 Title Bar Buttons and FrameStyle
- 6.13.2 Minimizing, Maximizing and Restoring the Form
- 6.13.3 Form's Position
- 6.13.4 Icon
- 6.13.5 Layout Manager
- 6.13.6 Showing and Closing the Form
- 6.13.7 Modal Forms
- 6.13.8 Asking User to Confirm the Closing of a Form
- 1 6. Creating Applications Using Components

# 6.13.1 Title Bar Buttons and FrameStyle

Use the BorderIcons property to define which buttons are shown in the title bar. By default the Minimize, Maximize and Close buttons are shown.



Any changes you make to these buttons are only shown at run-time, not at design-time.

You can select the Help button only if the Minimize or Maximize buttons are not selected.



At run-time when the Help button is clicked, the cursor is changed to contextsensitive help icon.



Then when the user clicks on a component on the form, context-sensitive help for that component is displayed if it exists.

In addition to the BorderIcons property, the FrameStyle property of the form affects the title bar buttons. This property also controls whether the window has a frame, title bar or an icon at run-time.

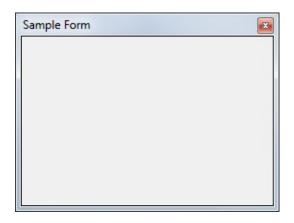
A form with FrameStyle Dialog is not sizable and it does not have a system menu in the left top corner. It can have Minimize, Maximize and Help buttons.



A form with FrameStyle Sizable has a system menu and can be sized using the mouse or the system menu:



A form with FrameStyle ToolWindow or SizeToolWindow has only the close button and the text of the title is in small font:



These forms cannot have an icon and they do not appear on the Windows task bar. Forms with FrameStyle SizeToolWindow are sizable.

## 6.13.2 Minimizing, Maximizing and Restoring the Form

You can minimize, maximize and restore the form programmatically using the MinimizeForm, MaximizeForm and RestoreForm methods.

This command restores the form to the size it was before it was maximized: invoke #formc.restoreform

To find out whether the form has been minimized, maximized or not, use the WindowState property of the form. The WindowState can be minimized, maximized or normal. This code shows the form's window state in a message box when the form is closed:

EVTROUTINE HANDLING(#COM\_OWNER.Closing) USE BUILTIN(OV\_MESSAGE\_BOX) WITH\_ARGS(#FORMC.WINDOW ENDROUTINE

# 6.13.3 Form's Position

Use the FormPosition property to specify the size and the position of the form at run-time. Any changes you make to the FormPosition values are only visible at run-time.

You can specify that at run-time the form has the same position and size as at design time (Designed), that it is centered or that it uses default position and/or size. The default size and position of new forms is specified in the #STD\_FORM component which you can modify.

### 6.13.4 Icon

Use the Icon property to specify a repository-enrolled icon to be used for your form. This icon is displayed in the top left corner of the form:



and on the task bar:



If you do not specify an icon, the default icon is used.

# 6.13.5 Layout Manager

Layout managers are used to control the positioning of components on a form regardless of the screen resolution. Refer to 6.29 What the Application will look like for more information.

# TitleBarVisible

You can use the TitleBarVisible property to display and hide the title bar of the form at run-time. When this property is set to False, the title bar is not visible, there is no system menu or border icons.

# 6.13.6 Showing and Closing the Form

A multi-form application is created by including forms in another form. The included forms are called member forms and the form in which they are included is called the owner form. For more information refer to 6.19 Multi-Form Applications.

To display a form from inside another form or reusable part, use the ShowForm method:

EVTROUTINE HANDLING(#PHBN\_1.Click) Invoke #FormA.ShowForm ENDROUTINE

If you want to make sure that a form that is displayed is brought to the front of all open windows use the ActivateForm event. To restore a minimized form, use the RestoreForm method.

To make sure a form is shown to the user regardless of whether it has been minimized, hidden by other forms or not displayed at all, you need to use all the three methods:

```
EVTROUTINE HANDLING(#PHBN_1.Click)
if cond('#Form1.windowstate *eq minimized')
Invoke #Form1.RestoreForm
Endif
Invoke #Form1.ShowForm
Invoke #Form1.ActivateForm
ENDROUTINE
```

# 6.13.7 Modal Forms

#### Sample Modal Form

If you want the form to be displayed modally, use the ShowModalForm method. A modal form requires the user to take some action before the form is closed. Typically you use modal forms when you want the user to perform an action before continuing.

EVTROUTINE HANDLING(#PHBN\_1.Click) Invoke #FormA.ShowModalForm ENDROUTINE

When the form is running, its ModalResult value is None. When the form is closed the ModalResult value is set to Cancel by default. You can set it to another value such as OK, Yes, No etc. This set of other values does not have predefined meanings.

When the ModalResult is any other value than None, the form is closed automatically.

You use the ModalResult property of buttons if you want to find out which button was pressed to close the form.

# Sample Modal Form

This example shows how a form is invoked modally and how the invoking form is notified which button on the modal form was pressed.

Our sample form which is to be invoked modally has two buttons, Button 1 and Button 2:

I			M	a	d	a		F	DT	m	1																C		2	×
1	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	·	•	•	•	•	•	•	•	•	•
:		:	:	:		:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:		:	:	:	:	:	
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
2					÷																÷									
•	•	•	•	•				D		H.,	or				Ł	•				D		н.	on	-	5		Ł	•	•	•
•	•	•	•	•				D	u	uu	Jr				Ł	•				D	u	uu	JE	1 4	-		Ł	•	•	•
•	•	•	•	•	1										-	•	-										•	•	•	•
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
1																														

The ModalResult value of Button 1 is set to Yes and the ModalResult value of Button 2 is set to No.

The form does not contain any event routines for the two buttons because when either button is clicked, the modal result is set to a value other than None which causes the form to close.

The form that will invoke the other form has a button used to launch the other form. It also has two radio buttons which indicate which button on the invoked form was clicked to close it.

3			В	ut	to	'n	1	v	Vā		F	br	e	55	e	d		-							·	•				Bı	ut	to	n	2		Na	15	p	ore	2	se	ed	-		2	3		
	•	•		•	•	•	•	•	•			•		•								•	•			• • •	•	•	•	•		•	•	•	•	•		•	•					 			)	
 	• • • •	•					•	• • • •	•					-		•	I	n	v	o	ke	e	N	10	od	la	F	Fo	or	m	۱		ļ	• • • • •	•	•	• • • • •	• • • • •	•					 				•
 • • • •	•	•		• • •	• • •	•				•	•	• • •	• • • •	• • • •	• • •	• • • •	• • • •					•			• • •	• • • •	•	• • • •	• • •	•	•	•	•			•	•							 • • • •	• • • •	• • •	• • •	•

The form to be invoked is included in this form. To do this drag-and-drop the modal form to the invoking form.

In the invoking form the Click event of the Invoke Modal Form button has logic

which sets the radio buttons according to the ModalResult value returned from the invoked form. The ModalResult is returned when the invoked form is closed.

```
EVTROUTINE HANDLING(#PHBN_1.Click)
*Invoke #MForm modally
invoke #MForm.ShowModalForm
*Test the returned ModalResult
IF COND('#Mform.ModalResult *eq Yes')
set com(#RDBN_1) ButtonChecked(True)
else
set com(#RDBN_2) ButtonChecked(True)
endif
ENDROUTINE
```

Both forms are saved and complied. The #INVFORM form is executed. When the Invoke Modal Form button is clicked, #MFORM is displayed. #MFORM is closed using either button and #INVFORM shows which button was clicked.

1	📜 Modal Form	
Invoking Form		
<ul> <li>Button 1 was pressed</li> </ul>		
Invoke M	Button 1	Button 2

Only push-buttons have the design-time property ModalResult. However, you can programmatically set the form's modal result based on other components. The following code assigns ModalResult Yes if radio button #RDBN\_1 is checked when the form is closed and to No if radio button #RDBN\_2 is checked when the form is closed:

EVTROUTINE HANDLING(#RDBN\_1.Click)

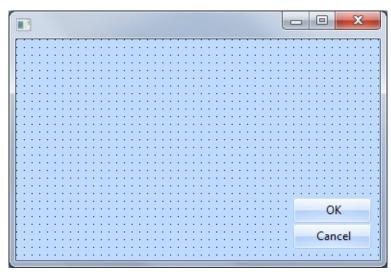
Set Com(#Com\_Owner) ModalResult(Yes) ENDROUTINE EVTROUTINE HANDLING(#RDBN\_2.Click) Set Com(#Com\_Owner) ModalResult(No) ENDROUTINE

1 6.13.7 Modal Forms

## 6.13.8 Asking User to Confirm the Closing of a Form

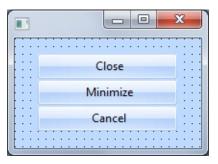
If you want to give the user the option of confirming whether a form should be closed, use the CloseFormQuery method and the CloseQuery event.

Our example has two forms: the main form and a dialog which is displayed when the form is about to be closed. The main form has two buttons, OK and Cancel.



The Cancel button closes the form using the CloseFormQuery method: Invoke #Com\_Owner.CloseFormQuery

In addition to the buttons, the form includes the Dialog form. The Dialog form has three buttons:



All the three buttons have a Click event routine which closes the form: EVTROUTINE HANDLING(#BTN\_CLOSE.Click) Invoke #Com\_Owner.CloseForm ENDROUTINE

#### EVTROUTINE HANDLING(#BTN\_CANCEL.Click) Invoke #Com\_Owner.CloseForm ENDROUTINE

EVTROUTINE HANDLING(#BTN\_MIN.Click) Invoke #Com\_Owner.CloseForm ENDROUTINE

In addition, they have their ModalResult values set as follows:

#BTN\_CLOSE Cancel
#BTN\_CANCEL No
#BTN\_MIN OK

In addition to the Cancel button event routine, the main form has an event routine for the CloseQuery event. This routine invokes the Dialog form and then receives its ModalResult when the Dialog form is closed. It then tests for the value of the returned ModalResult and performs actions accordingly. Note that whether the Closing of the form is continued depends on the value of

the #Option field which is used for the Continue parameter.

EvtRoutine Handling(#Com\_Owner.CloseQuery) Continue(#Option) Invoke #Dialog.ShowModalForm

Define #QResult Reffld(#Std\_Texts) Change #QResult #Dialog.modalResult

```
Case #QResult
When '= No'
Invoke #Com_Owner.RestoreForm
Set #Option Value(FALSE)
When '= OK'
Invoke #Com_Owner.MinimizeForm
Set #Option Value(FALSE)
OtherWise
Set #Option Value(TRUE)
EndCase
EndRoutine
```

The case statement restores the main form if the ModalResult value returned from the Dialog form is No (assigned to the Cancel button) and minimizes the main form if the value returned is OK (assigned to the Minimize button). In both cases the Continue parameter is set to False. If any other ModalResult value is returned, the form is closed.

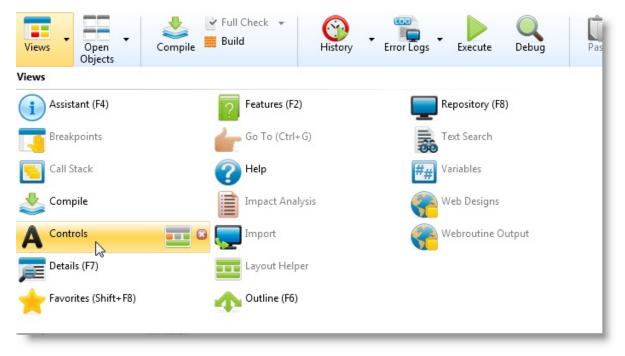
sample	form 🗆 🔍 🗙	Ŋ	
	Close		
	Minimize		
	Cancel		
		J (	ОК
		i i	Cancel

**↑** 6.13 Forms

# 6.14 Controls

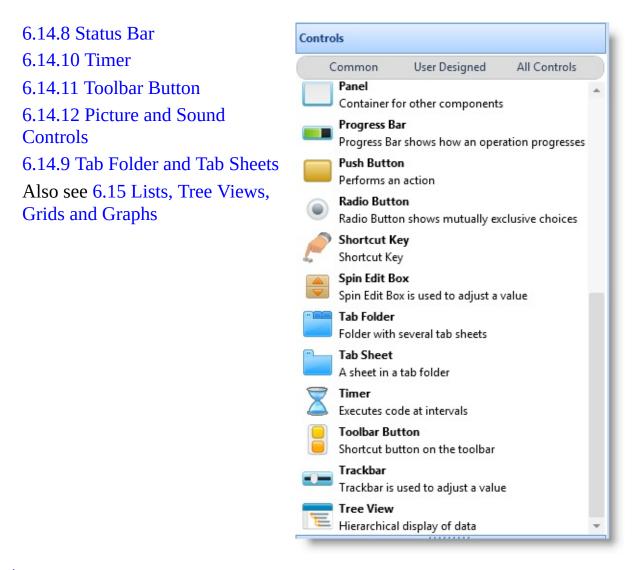
This section describes basic Visual LANSA controls. The controls are components which the user can use to control the application.

To display the *Controls* tab, use the *Views* command on the *Home* tab of the ribbon:



The use of most Visual LANSA controls is straightforward. For example, if you want to use an image in your program, drag the image control to your form and specify the file name of the picture. However, some controls are more complex. This section provides an introduction to these controls. For more detailed information, use F1 to access the Features Help for the control. The Features help describes the component which is currently selected in the editor.

6.14.1 Check Box
6.14.2 Group Box
6.14.3 Label
6.14.4 Panel
6.14.5 Push-Button
6.14.6 Radio Button
6.14.7 Static



1 6. Creating Applications Using Components

### 6.14.1 Check Box

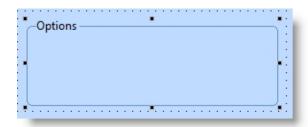


A check box is used to represent a Yes/No choice. The ButtonState property of the check box control is used to indicate whether a check box has been checked. Note that check box is also one of the available visualizations for a field. Before you use the check box control, you should aways consider whether using a field with this visualization would be a more appropriate choice in terms of reuse and consistency.

EVTROUTINE HANDLING(#PHBN\_1.Click) If cond '#ckbx\_1.ButtonState \*eq Checked' use message\_box\_show with\_args(OK OK Information ButtonState 'The box is checked') else use message\_box\_show with\_args(OK OK Information ButtonState 'The box is not checked') endif ENDROUTINE

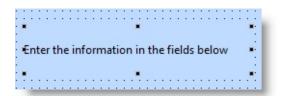
A check box is also a visualization style for a field.

## 6.14.2 Group Box



A group box provides a grouping for other components. A group box can optionally have a title. Typically group boxes are used to group together a set of check boxes or radio buttons. In the case of radio buttons, only one button can be selected at a time inside the box.

## 6.14.3 Label



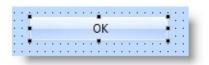
Use a label for text the user cannot change. Labels are usually used for instructions and together with an edit box. You enter the text for the label in its Caption property.

# 6.14.4 Panel

																											1
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
•																										•	•
•																										•	1
•																										•	1
•																										•	•
•																										•	•
																											1
•	-												-												-		1
٠	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	

A panel provides a visual grouping of items in a similar way as a group box, but without a title.

### 6.14.5 Push-Button



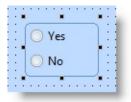
A push-button is used to allow the user to perform an action. The code for the button is placed in its Click event.

A push-button can have a bitmap or a text caption or both.

The following code updates fields in the PSLMST file when a button (#PHBN\_1) is clicked:

```
EVTROUTINE HANDLING(#PHBN_1.Click)
update fields(#givename #surname) in_file(pslmst)
ENDROUTINE
```

### 6.14.6 Radio Button



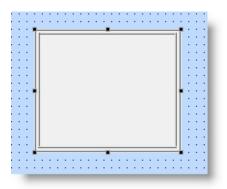
Radio buttons are used to represent mutually exclusive choices. The convention is to enclose the radio buttons in a group box.

The ButtonChecked property of the radio button indicates whether it is checked. By default the buttons are not checked. The following code issues a message telling which one of two radio buttons has been checked:

```
EVTROUTINE HANDLING(#PHBN_1.Click)
If cond('#rdbn_1.ButtonChecked *EQ true ')
use builtin(message_box_show) with_args(OK OK Information 'Radio Butt
'Radio button 1 has been selected')
else
use builtin(message_box_show) with_args(OK OK Information 'Radio Butt
'Radio button 2 has been selected')
endif
ENDROUTINE
```

A field can be visualized as a set of radio buttons.

### 6.14.7 Static



### 6.14.8 Status Bar



A status bar creates a separate area at the bottom of the window where messages are displayed. The status area can also contain other components, such as progress bars, (animated) pictures and edit boxes. You determine the relative position of the components in the status bar using the DisplayPosition and Message position properties.

You can display status and information messages in the status bar. The status messages appear and disappear as an operation progresses, whereas information messages are placed in a queue so that if there is more than one message, they can be scrolled through.

The easiest way of using a status bar is to run the VL\_BBSTSBR template which places the status bar on the form and creates two subroutines: infmessage and stsmessage. The infmessage subroutine is used like this:

```
execute infmessage ('Processing Completed')
```

and the stsmessage subroutine syntax is:

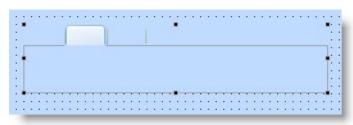
```
execute stsmessage (LOOPS #LOOPCOUNT MESSAGE)
```

where LOOPS sets the number of loops after which the message is issued, #LOOPCOUNT is the number of the current loop and MESSAGE is the message text. The following code would issue a message 'nnnn Loops' in the status bar after every 1000 loops:

```
change #loopcount 0
begin_loop from(1) to(10000)
execute stsmessage (1000 #loopcount 'Loops')
end_loop
```

Note that validation messages issued by I/O modules are displayed automatically if the form has a status bar.

## 6.14.9 Tab Folder and Tab Sheets



To add a tab folder with sheets on a form, first drag tab folder to the form and then as many sheets as you want to the folder. Alternatively you can run the VL\_BASETAB template (choose the Templates command on the Design tab of the ribbon) to create a tab folder with four sheets.

Tab sheets have a run-time property Opened, which indicates whether it is currently the topmost tab in the folder and two related events: Opening and Closing. Opening event is triggered when the user has clicked on a tab sheet to bring it to front, and Closing is triggered when another tab is being opened.

## 6.14.10 Timer

A timer component is used to execute code after a certain time has elapsed or at specified intervals. The code to be executed is written in the Tick event of the timer.

You can use it, for example, to show and hide an image every time after a certain number of milliseconds have elapsed. The number of milliseconds is set in the Interval property of the timer. You could also use a timer to display a message or end an application after a set period of time.

The timer has a Start and a Stop method:

```
EVTROUTINE HANDLING(#PHBN_1.Click)
Invoke #TIMR_1.Start
ENDROUTINE
```

The following code alternately displays and hides an image at every tick of the timer:

```
EVTROUTINE HANDLING( #TIMR_1.Tick )
if '#imge_1.Visible *EQ true'
set #imge_1 visible(false)
else
set #imge_1 visible(true)
endif
ENDROUTINE
```

### 6.14.11 Toolbar Button



Toolbar buttons are used to allow the user to quickly perform an action. There should be a toolbar button for the most commonly used actions in the application.

To start creating a toolbar, drag a panel component (PRIM\_PANL) to the form and size it. Usually a toolbar panel is placed underneath the form's menu bar. If you want to group toolbar buttons, you can add other panels to the toolbar panel.

To create the toolbar, drag toolbar buttons to the panel. Use the Image property of a button to specify a repository-enrolled bitmap to be displayed on it. Note that you cannot specify text on a toolbar button, but you should use the Hint property to add a ToolTip description for it. The size of a toolbar button is adjusted to the size of the bitmap on it. Typically bitmaps used on toolbar buttons are 16 x 16 pixels.

## 6.14.12 Picture and Sound Controls

The purpose of image and sound controls is to show an image or a video file or to play a sound file.

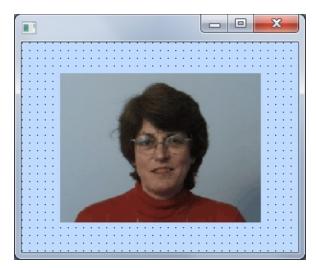
### Image



Use the image control to display an image file. When the image you want to display is not enrolled in the repository, use the FileName property to display the name of the image file and when you want to display a repository-enrolled bitmap, use the Image property. Images that rarely change should be stored in the repository so that they are easier to find and manage. The image control can also display animations (animated .gif files).

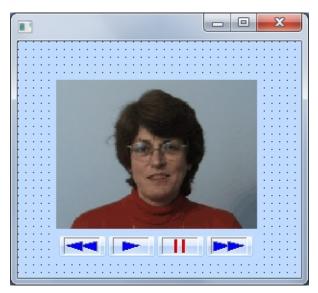
The Visual LANSA installation CD has a directory VLRES that contains .gif and .bmp files you can use. You can also copy any image on the Internet by selecting it, right-clicking to bring up a pop-up menu and using the Save Image As... option. However, if you are using these images in commercial applications, make sure you are not infringing any copyrights.

### Video



Use the video control to display a video file. The name of the video file is specified using the FileName property.

## Video Panel



Use a video panel control to display a video file and to let the user play, pause and wind the video.

### Sound

Use a sound control to play a sound file.

# Specifying the File Name for Pictures and Sound

The image, video or sound file to be displayed in a control is specified using the FileName property. You can, of course, specify the full path where the file is stored, but this can easily cause problems when running the application on another workstation.

A better way of specifying the file name is to use a system variable for the path name. For example you could create a variable \*VIDEOPATH and set its value to the path where you have stored the videos. You could then change the value of the variable as required:

```
define field(#FileName) type(*char) length(70)
use tconcat (*Videopath MyVideo.avi) (#FileName)
set #video_1 filename(#FileName)
```

Alternatively you could use one of the standard LANSA system variables such as \*SYS\_DIR to specify the path. For example if the file was installed in a directory PICTURES in the LANSA root directory (x:\X\_Win95\X\_LANSA), you could specify:

```
use tconcat (*SYS_DIR PICTURES\MyVideo.avi) (#FileName)
set #video_1 filename(#FileName)
```

## 6.15 Lists, Tree Views, Grids and Graphs

In Visual LANSA there are several controls which offer different ways of retrieving, displaying and maintaining list-type information. All lists show the values for one or more LANSA fields.

Most of these controls are straightforward lists or tables: list box, list view, combo box and grid. Tree views are a special kind of list which show hierarchical information. Two others, graphs and multi-line edit boxes, do not appear to be lists to the user, but they operate the same way as lists.

Essentially, all list type components are containers for fields. To create a list, you first add the list component to your form and then you add to it the fields you want displayed. Every field you add becomes a column in the list.

- 6.15.1 Combo Box
  6.15.2 Grid
  6.15.3 List Box
  6.15.4 Tree View
  6.15.5 List View
  6.15.6 Defining a List-Type Component
  6.15.7 Filling a List-Type Component
  6.15.8 Working with Items in Lists
- ↑ 6. Creating Applications Using Components

## 6.15.1 Combo Box

Use a combo box to present a list from which the user can select entries. A combo box consists of an edit box portion and a list portion. The ComboBoxStyle property determines the style of the combo:

#### Simple

Black	
Black	
Blake	
Bloggs	
Brown	
Brown	

In a simple combo box both the edit box and the list portion are shown.

#### **Drop-Down**



In a drop-down combo box the list portion is hidden until the user clicks on the spin button. This style is useful when you want to fit a list in a small place.

#### **Drop-Down List**



A drop-down list is otherwise the same as a drop-down, except it does not allow the user to enter data.

### ↑ 6.15 Lists, Tree Views, Grids and Graphs

## 6.15.2 Grid

Employee Number	Employee Surname	Employee Given Name(s)	
A0070	Brown	Veronica	
A0090	Bloggs	Fred	6
A0193	Smithson	Fred	
A0907	Simpson	Anne	
A1001	Jones	Shirley	
A1002	Smythe	John	Ŧ
4		F I I I I I I I I I I I I I I I I I I I	

Use a grid to display data in table format. The grid allows the user to easily work with data in several columns (fields):

- The cells in the grid can be manipulated independently.
- The user can resize and reorganize columns and adjust the height of individual rows.
- For quick data entry the Enter and Cancel buttons can be used to move focus to the next cell and to reset values.
- When column headings are used, the user can sort the entries in a column in ascending or descending order.
- A large grid can be split into several panes to be viewed independently.

↑ 6.15 Lists, Tree Views, Grids and Graphs

## 6.15.3 List Box

Brown	Veroni	
Bloggs	Fred	
Smithson	Fred	
Simpson	Anne	
Jones	Shirley	-

List box is a basic list. It has limited functionality compared to the list view control.

↑ 6.15 Lists, Tree Views, Grids and Graphs

## 6.15.4 Tree View

Use a tree view to show hierarchical data.



A tree view is a good way of presenting, for example:

- Files grouped by directories
- Employees grouped by departments
- Orders grouped by customers and customers grouped by area

You can nest several levels of information, but in most cases it is clearest not to use more than two or three levels.

The images in the tree view are defined using the Image and ImageExpanded properties of the individual tree view columns.

**Populating Tree Views** 

Filling a Drill-Down Tree-View

Filling a Complete Tree View

Filling a Tree View on Demand

Level and KeyPosition in Tree Views

1 6.15 Lists, Tree Views, Grids and Graphs

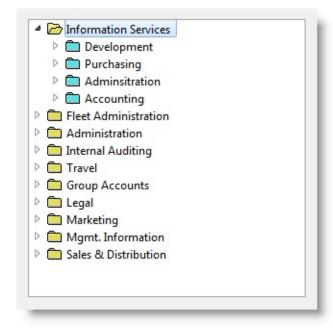
## **Populating Tree Views**

Populating tree views is slightly different because the information is nested in levels inside the tree. There are essentially three ways of filling a tree:

- All levels in one go as a drill-down list. A drill-down list only includes items that contain other items. For example if you had a tree view showing employees by department, a department with no employees would not be shown. See Filling a Drill-Down Tree-View.
- All levels in one go. See Filling a Complete Tree View.
- Every level on demand. See Filling a Tree View on Demand.
- ↑ 6.15.4 Tree View

## Filling a Drill-Down Tree-View

A drill-down tree only includes items that contain other items. To fill a drill down tree, use a single SELECT statement to retrieve all the fields. Data from other files is fetched as required. This example fills a three-level tree view of departments, sections, and employees from the PSLMST file. It shows all sections that contain employees and all departments which contain sections.



To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
```

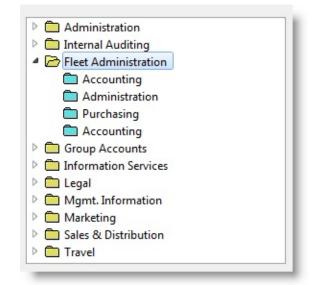
BEGIN\_COM role(\*EXTENDS #PRIM\_FORM) FORMPOSITION(ScreenCe DEFINE\_COM class(#PRIM\_TRVW) name(#TRVW\_1) DISPLAYPOSITION DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_1) KEYPOSITION(1) L1 DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_3) KEYPOSITION(1) L1 DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_2) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_4) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_5) KEYPOSITION(1) L1 DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_6) DISPLAYPOSITION(1) L1

EVTROUTINE handling(#TRVW\_1.Initialize) options(\*NOCLEARMESSAC SELECT fields(#EMPNO #SURNAME #DEPTMENT #SECTION) from\_file FETCH fields(#DEPTDESC) from\_file(DEPTAB) with\_key(#DEPTMENT) FETCH fields(#SECDESC) from\_file(SECTAB) with\_key(#DEPTMENT #SE ADD\_ENTRY to\_list(#TRVW\_1) ENDSELECT ENDROUTINE END\_COM

↑ 6.15.4 Tree View

## Filling a Complete Tree View

To fill in a tree view so that also items which do not include any items are included, use nested SELECT statements. This example shows all departments and sections in the PSLMST file:



To see how the example works, copy this code and paste it to a form component:

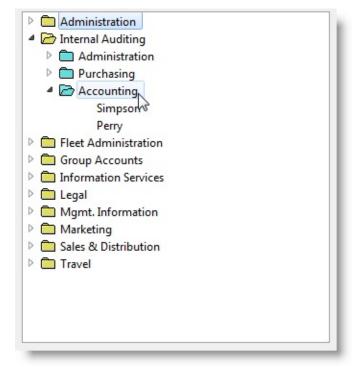
```
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(331) LEFT(334) T
DEFINE_COM class(#PRIM_TRVW) name(#TRVW_1) DISPLAYPOSITION
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_1) KEYPOSITION(1) L1
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_3) KEYPOSITION(1) L1
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_4) DISPLAYPOSITION(
```

```
EVTROUTINE handling(#TRVW_1.Initialize) options(*NOCLEARMESSAC
SELECT fields(#DEPTMENT #DEPTDESC) from_file(DEPTAB)
ADD_ENTRY to_list(#TRVW_1)
SELECT fields(#SECTION #SECDESC) from_file(SECTAB) with_key(#DEI
ADD_ENTRY to_list(#TRVW_1)
ENDSELECT
ENDSELECT
ENDROUTINE
END_COM
```

↑ 6.15.4 Tree View

## Filling a Tree View on Demand

To fill in a tree view only when the user double-clicks on items, you need to test what level of the tree is being worked with and fetch the data in the ItemExpanding event in the tree view. (See the next section for information about the tree view properties). This example retrieves the section and the employee information on demand:



When you use a dynamic tree view, set the ManageChildren property of the tree view to True.

To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
```

```
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(405) LEFT(129) T
DEFINE_COM class(#PRIM_TRVW) name(#TRVW_1) DISPLAYPOSITION
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_1) KEYPOSITION(1) LI
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_3) KEYPOSITION(1) LI
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_4) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_5) KEYPOSITION(1) LI
DEFINE_COM class(#PRIM_TVCL) name(#TVCL_6) DISPLAYPOSITION(1) LI
```

EVTROUTINE handling(#TRVW\_1.Initialize) options(\*NOCLEARMESSAC SELECT fields(#DEPTMENT #DEPTDESC) from\_file(DEPTAB) ADD\_ENTRY to\_list(#TRVW\_1) ENDSELECT ENDROUTINE

EVTROUTINE handling(#TRVW\_1.ItemExpanding) CHANGE field(#STD\_NUM) to('#TRVW\_1.CURRENTITEM.LEVEL') CASE of\_field(#STD\_NUM) WHEN value\_is('= 1') SELECT fields(#SECTION #SECDESC) from\_file(SECTAB) with\_key(#DEI ADD\_ENTRY to\_list(#TRVW\_1) ENDSELECT WHEN value\_is('= 2') SELECT fields(#EMPNO #SURNAME) from\_file(PSLMST1) with\_key(#DE ADD\_ENTRY to\_list(#TRVW\_1) ENDSELECT ENDCASE ENDROUTINE

END\_COM

↑ 6.15.4 Tree View

## Level and KeyPosition in Tree Views

A tree view which is a hierarchical list has two properties other lists do not have: Level and KeyPosition. Level specifies which level in the hierarchy the column belongs to and KeyPosition specifies whether the column is used as a key for this level of the list. Every level in a tree view except the last must have a field with a KeyPosition specified.

For example a value of 1 for Level specifies that the column is shown on the first level of the tree view. A value of 0 for KeyPosition specifies that the column is not used as the key for this level. There can be more than one key column for a level in which case the KeyPosition determines their order. Note however that only one column can be visible per level.

↑ 6.15.4 Tree View

#### 6.15.5 List View

Employee Given Name(s)	Employee Surname	
Veronica	Brown	
Fred	Bloggs	
Fred N	Smithson	
Anne 😼	Simpson	
Shirley	Jones	
John	Smythe	
Robert	Smithe	
Ruth	Smithson	
Peter	Smiths	
Jack	Smithers	*

Use a list view to offer the user various ways of displaying a list. You can arrange items into columns with or without column headings as well as display images for the list view items.

A list view can be displayed in different views using the ViewStyle property: Icons, Small Icons, List, Report.

You can add checkboxes to the list view by setting the CheckBoxes property of the list view to True:

Employee Given Name(s)	Employee Surname	
Veronica	Brown	-
Fred	Bloggs	
√ Fred	Smithson	
Mnne	Simpson	
Shirley	Jones	
John	Smythe	
Robert	Smithe	
Ruth	Smithson	-

You can let the user sort the items in a list view column by clicking on the column heading if you set the SortOnClick property of the column to True. The sort sequence swaps between descending and ascending.

Employee Given Name(s)	Employee Surname	-
Paul	Zacharia 😽	
Bradley	Woods	
Warren	Verey	
Valerie	Turner	
Jack	Turner	
Kelly	Thompson	
John	Taylor	
George	Snell	
Alison	Sneddon	
Damian	Snashall	*

To assign an image to a list view item use the SET command to assign an image to the current item in the list:

EVTROUTINE handling(#COM\_OWNER.Initialize) options(\*NOCLEARME SELECT fields(#LTVW\_1) from\_file(PSLMST) ADD\_ENTRY to\_list(#LTVW\_1) SET com(#LTVW\_1.CURRENTITEM) IMAGE(#VI\_EXEMPL) ENDSELECT ENDROUTINE

EVTROUTINE handling(#ltvw\_1.itemgotselection) SET com(#LTVW\_1.CURRENTITEM) IMAGE(#VI\_EMPLOY) ENDROUTINE

You can also assign an image to the list view columns by using the Image property of the column:

🜇 Employee Nu	Employee Surname	Employee Given Nam	
A0070	Brown	Veronica	
A0090	Bloggs	Fred	
A0193	Smithson	Fred	
A0907	Simpson	Anne	
A1001	Jones	Shirley	
A1002	Smythe	John	
A1003	Smithe	Robert	
A1004	Smithson	Ruth	
A1005	Smiths	Peter	
A1006	Smithers	Jack	*

6.15.6 Defining a List-Type Component6.15.7 Filling a List-Type Component6.15.8 Working with Items in Lists

1 6.15 Lists, Tree Views, Grids and Graphs

## 6.15.6 Defining a List-Type Component

You use repository fields to specify what data is to be displayed in the list. To define what fields the list is to contain, drag them from the repository to the list.

As you can see, every field you drag to the list becomes a column in the list. By default the columns are named xxCL\_n, for example GDCL\_1 for the first grid column. You can see the column definition in the Source view of the editor and you can view the column's properties in the Details tab. Note that the column definition shows the list to which it belongs (#Grid\_1 in the next picture) and the field from which it is sourced (#SKILCODE).

Details		
GDCL_1		*
Ē ▼ ?		
Properties Events Methods		
ComponentTag		
ComponentTypeName	PRIM_GDCL	
DisplayAppearance	Default	
DisplayPosition	1	
EditAppearance	Default	
🖻 EditorPart	*NULL	
P Hint		
🖻 Image	*NULL	
MinimumWidth	0	
P Name	GDCL_1	
🖉 Owner	#DEMCOM_1	
Parent 🖉	#GRID_1	
🖻 PopupMenu	*NULL	
🖻 ReadOnly	True	
🖻 RowAlign	Center	
SelectOnClick	False	=
🖉 SortAsColumn	*NULL	
SortDirection	Ascending	
SortOnClick	False	
SortPosition	0	
SortType	Word	
P Source	#SKILCODE	
🖉 TabStop	True	
UsePicklist	False	
🖉 Visible	True	
🖉 VisualStyle	*NULL	
P Width	11	-

This is a five-column grid:

:						
•	Skill Code	Skill Full Description	Date Skill Acqui	Grade Obtained for Skill	Comment on skills ac	:
:	ABCDEFG	aAbBcCdDeEfFgGhHi	12/34/56	A	aAbBcCdDeEfFgGhH	:
:						:
						•
•						÷
						÷
ł						

Its definition is created automatically as the fields are dragged to the grid:

DEFINE\_COM class(#PRIM\_GRID) name(#GRID\_1) COLUMNBUTTONH DEFINE\_COM class(#PRIM\_GDCL) name(#GDCL\_1) CAPTIONALIGN(L¢ DEFINE\_COM class(#PRIM\_GDCL) name(#GDCL\_2) CAPTIONALIGN(L¢ DEFINE\_COM class(#PRIM\_GDCL) name(#GDCL\_3) CAPTIONALIGN(L¢ DEFINE\_COM class(#PRIM\_GDCL) name(#GDCL\_4) CAPTIONALIGN(L¢ DEFINE\_COM class(#PRIM\_GDCL) name(#GDCL\_5) CAPTIONALIGN(L¢

Hidden Columns
Specifying the Column Width
Sorting the Lists
↑ 6.15 Lists, Tree Views, Grids and Graphs

## **Hidden Columns**

You can define whether a column is visible using its DisplayPosition property. When this property is set to 0, the column is hidden.

Very often you would include a column such as skill code, employee number or invoice number as the column which identifies the entries in this list and hide it. In this way you can use a unique identifier for the entries in reading and writing to the database while at the same time only displaying information that is relevant to the user.

↑ 6.15.6 Defining a List-Type Component

## Specifying the Column Width

The recommended way of specifying column widths is to set the WidthType to Scaleable which allows the column widths to be specified as a percentage of the width of the entire list. So for example a column with a Width of 20 would take up one fifth of the entire list width. The last column in a list should be specified as WidthType Remainder in which case the column takes up whatever space is left unused.

The width of the columns can be adjusted visually by sizing the column headers in the design view. The value of the width property is adjusted accordingly.

↑ 6.15 Lists, Tree Views, Grids and Graphs

## Sorting the Lists

Lists have two properties which determine how the entries are sorted: SortOrder and SortPosition. SortOrder is by default Ascending. The SortPosition property determines the position of the column in the sort order. A column with a SortOrder of 1 is the primary column which is used to sort the list. For instance if you had a three-column list with Employee Number sort order 0, Last Name sort order 1 and First Name sort order 2, the employee number would be ignored in the sort order and the entries would be sorted by last name and then by first name.

↑ 6.15 Lists, Tree Views, Grids and Graphs

## 6.15.7 Filling a List-Type Component

```
All lists are populated using a SELECT loop with an ADD_ENTRY statement:
SELECT fields(#EMPLIST) from_file(PSLMST)
ADD_ENTRY to_list(#EMPLIST)
ENDSELECT
```

In this example the list view is filled when the Fill List button is clicked:

Fill List				
Employee Num	Employee Surna	Employee Given	Employee Salary	T
A0070	Brown	Veronica	50,125.00	
A0090	Bloggs	Fred	20,045.91	
A0193	Smithson	Fred	35,000.04	
A0907	Simpson	Anne	34,213.04	
A1001	Jones	Shirley	2,345.82	
A1002	Smythe	John	25,000.04	
A1003	Smithe	Robert	31,000.04	
A1004	Smithson	Ruth	21,000.04	
A1005	Smiths	Peter	46,700.04	
A1006	Smithers	Jack	25,000.04	
A1007	Snell	George	26,780.04	
A1008	Sneddon	Alison	450,000.04	-

To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
BEGIN_COM ROLE(*EXTENDS #PRIM_FORM) CLIENTHEIGHT(295)
CLIENTWIDTH(480) HEIGHT(333) LEFT(537) TOP(173) WIDTH(496)
DEFINE_COM CLASS(#PRIM_LTVW) NAME(#EMPLIST)
COLUMNBUTTONHEIGHT(16) DISPLAYPOSITION(1)
FULLROWSELECT(True) HEIGHT(209) LEFT(24)
PARENT(#COM_OWNER) TABPOSITION(1) TOP(48) WIDTH(433)
DEFINE_COM CLASS(#PRIM_LVCL) NAME(#LVCL_1)
DISPLAYPOSITION(1) PARENT(#EMPLIST) SOURCE(#EMPNO)
WIDTH(24)
DEFINE_COM CLASS(#PRIM_LVCL) NAME(#LVCL_2)
DISPLAYPOSITION(2) PARENT(#EMPLIST) SOURCE(#SURNAME)
```

WIDTH(25)

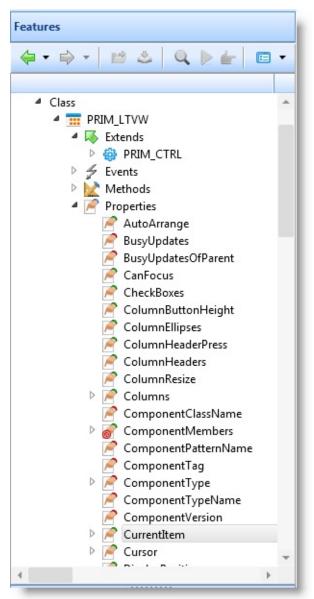
DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_3) DISPLAYPOSITION(3) PARENT(#EMPLIST) SOURCE(#GIVENAME) WIDTH(25) DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_4) DISPLAYPOSITION(4) PARENT(#EMPLIST) SOURCE(#SALARY) WIDTH(21) WIDTHTYPE(Remainder) DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#PB\_FILL) CAPTION('Fill List') DISPLAYPOSITION(2) LEFT(32) PARENT(#COM\_OWNER) TABPOSITION(2) TOP(8) EVTROUTINE handling(#PB\_FILL.Click) SELECT fields(#EMPLIST) from\_file(PSLMST) ADD\_ENTRY to\_list(#EMPLIST) ENDSELECT ENDROUTINE ENDCOM

↑ 6.15 Lists, Tree Views, Grids and Graphs

## 6.15.8 Working with Items in Lists

When an application is running, all list items have at least two properties: Selected and Focus. These properties can be set to True or False. For example if an item does not have the focus nor is selected, both these properties are set to False.

The item properties are manipulated using the list's CurrentItem or FocusItem properties.



The CurrentItem is the item in a list box, list view, combo box or grid which is currently being worked with in the application. The FocusItem is the item in a

list, tree view or grid which has the focus on the desktop.

CurrentItem can be set by the user by moving the focus on the item (ItemGotFocus event), in which case it is the same as FocusItem, or it can be set by the program in a GET\_ENTRY or SELECTLIST command.

The value of the current item is used as the key in database I/O commands. Therefore data is typically fetched in the ItemGotFocus event which sets the CurrentItem.

You can use the SELECTLIST command to examine the properties of every item in a list. For example the following code goes through the list making one item at a time the CurrentItem and tests whether its Selected property is set to true:

```
SELECTLIST NAMED(#LTVW_1)
IF '#ltvw_1.CurrentItem.Selected *eq True'
do something
ENDIF
ENDSELECT
```

In lists where it is not possible to select more than one entry at a time (tree view, combo box, and list box), the Focus and Selected properties have always the same value. In other words only the item which has focus can be selected. In lists which allow for multiple selection, only one item can have the focus (Focus property set to True), but multiple items can be selected (usually by using the Ctrl and Shift keys).

You can test for the Focus property of items the same way as you test the Selected property:

#ltvw\_1.CurrentItem.Focus \*eq True

#### See also

Understanding selection and loss of selection in list views	Dynamically Assigning Icons in a List View
Understanding focus and loss of focus in list views	Creating a Desktop Menu Using Large Icons in List View
Run-time Properties of Tree Items	Sorting List Views 1
The Value of a List Item Is the Value	Sorting List Views 2
of the Source Field	Saving and Restoring a List from a

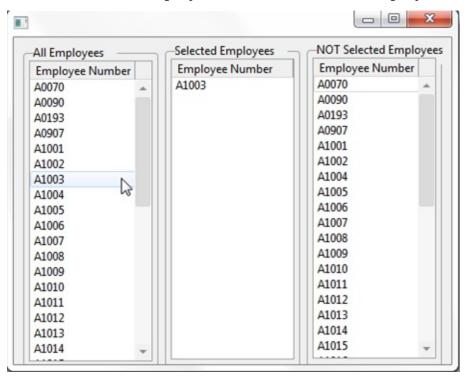
Updating the Selected Item in a List Updating All Items in a List Deleting an Item from a List

File Special Lists: Graph Special Lists: Multi-line Edit Box Error Handling in a Grid/List

1 6. Creating Applications Using Components

#### Understanding selection and loss of selection in list views

This example uses three list views. At start up all employees are added to the All Employees and Not Selected Employees list views. As employees are selected and deselected from the All Employees list they are added to and removed from the Selected Employees and Not Selected Employees list.



To see how the example works, copy this code and paste it to a form component:

#### FUNCTION options(\*DIRECT)

```
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(378) LEFT(345) T
DEFINE_COM class(#PRIM_LTVW) name(#ALL_LIST) DISPLAYPOSITIC
DEFINE_COM class(#PRIM_LTVW) name(#SEL_LIST) DISPLAYPOSITIO
DEFINE_COM class(#PRIM_LTVW) name(#NOT_LIST) DISPLAYPOSITIC
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_1) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_3) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#GPBX_ALL) CAPTION('All Er
DEFINE_COM class(#PRIM_GPBX) name(#GPBX_SEL) CAPTION('Select
DEFINE_COM class(#PRIM_GPBX) name(#GPBX_NOT) CAPTION('NOT
DEFINE_COM class(#PRIM_GPBX) name(#GPBX_NOT) CAPTION('NOT
DEFINE field(#SEL_EMPNO) reffld(#EMPNO)
EVTROUTINE handling(#com_owner.Initialize)
```

\* Add all employees to the all and not selected lists SELECT fields(#ALL\_LIST) from\_file(PSLMST) ADD\_ENTRY to\_list(#ALL\_LIST) ADD ENTRY to list(#NOT LIST) **ENDSELECT ENDROUTINE** EVTROUTINE handling(#ALL\_List.ItemGotSelection) \* Save employee number value CHANGE field(#SEL\_EMPNO) to(#EMPNO) \* Add to Selected list ADD\_ENTRY to\_list(#SEL\_LIST) \* Remove from the not selected list SELECTLIST named(#NOT\_LIST) IF cond('#EMPNO = #SEL EMPNO') DLT\_ENTRY from\_list(#NOT\_LIST) LEAVE **ENDIF ENDSELECT** ENDROUTINE EVTROUTINE handling(#ALL\_List.ItemLostSelection) \* Save employee number value CHANGE field(#SEL\_EMPNO) to(#EMPNO) \* Add to not Selected list ADD\_ENTRY to\_list(#NOT\_LIST) \* Remove from the selected list SELECTLIST named(#SEL LIST) IF cond('#EMPNO = #SEL EMPNO') DLT ENTRY from list(#SEL LIST) LEAVE **ENDIF ENDSELECT ENDROUTINE** END COM

## Understanding focus and loss of focus in list views

This example fills a list view with employee details. Additionally the list view contains two columns titled Selected and Focused that contain YES when the employee list view entry is selected or in focus.

Focus	Selected	Employee Number	Employee Surname	Employee Given Name(s)
		A0070	Brown	Veronica
		A0090	Bloggs	Fred
		A0193	Smithson	Fred
		A0907	Simpson	Anne
	YES	A1001	Jones	Shirley
	YES	A1002	Smythe	John
	YES	A1003	Smithe	Robert
YES	YES N	A1004	Smithson	Ruth
	2	A1005	Smiths	Peter
		A1006	Smithers	Jack
		A1007	Snell	George
		A1008	Sneddon	Alison
		A1009	Snashall	Damian
		A1010	Perry	Kate
		A1011	Perron	Christopher

To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
```

```
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(346) LEFT(381) T
DEFINE_COM class(#PRIM_LTVW) name(#EMPLIST) DISPLAYPOSITIO
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_1) CAPTION('Focus') CA
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_2) CAPTION('Selected')
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_3) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_4) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_5) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_5) DISPLAYPOSITION(
EVTROUTINE handling(#com_owner.Initialize)
SELECT fields(#EMPLIST) from_file(PSLMST)
CHANGE field(#STD_CODEL #STD_CODE) to(*BLANKS)
ADD_ENTRY to_list(#EMPLIST)
ENDSELECT
ENDROUTINE
EVTROUTINE handling(#EMPLIST.ItemGotFocus)
```

CHANGE field(#STD\_CODE) to(YES) UPD\_ENTRY in\_list(#EMPLIST) **ENDROUTINE** EVTROUTINE handling(#EMPLIST.ItemLostFocus) CHANGE field(#STD\_CODE) to(\*BLANKS) UPD\_ENTRY in\_list(#EMPLIST) **ENDROUTINE** EVTROUTINE handling(#EMPLIST.ItemGotSelection) CHANGE field(#STD\_CODEL) to(YES) UPD\_ENTRY **ENDROUTINE** EVTROUTINE handling(#EMPLIST.ItemLostSelection) CHANGE field(#STD\_CODEL) to(\*BLANKS) UPD\_ENTRY in\_list(#EMPLIST) **ENDROUTINE** END\_COM

## **Run-time Properties of Tree Items**

In a tree view, the items can have additional properties: Bold, HasChildren, Level and Expanded.

- The Bold property makes the font of the caption of the item in bold. It can be set to True or False
- The HasChildren property indicates whether the item contains child items. Its value can be Yes, No or Unknown.
- The Level property is read-only. It indicates on what level of the tree view the focus item is. Its value is a number. See the help for the ItemExpanding event to see an example of how to use this property.
- The Expanded property is used check and to control whether a tree view item which has children (called a node) is expanded or collapsed. It can have a value of True or False.

This command makes the caption of the current item bold:

set com(#trvw\_1.CurrentItem) Bold(True)

### The Value of a List Item Is the Value of the Source Field

When you want to find out the value of a list item, use the field from which the list column is sourced (displayed as the Source property of the Column).

The following example uses a tree column sourced from the #SURNAME field. This field is used to find out the value of surname for the current item in the tree and then to set the item with surname BROWN to bold

IF COND'#SURNAME \*EQ BROWN' SET COM(#TRVW\_1.CURRENTITEM) BOLD(TRUE) ENDIF

## Updating the Selected Item in a List

To update the selected item in a list use the UPD\_ENTRY command.

In this example an employee must be selected to enable the "Give Raise" push button. The salary increase is only applied to the selected employee.

Employee Number	Employee Surname	Employee Given Name(s)	Employee Sal
A0070	Brown	Veronica	50,125.00
A0090	Bloggs	Fred	20,045.91
A0193	Smithson	Fred	35,000.04
A0907	Simpson	Anne	34,213.04
A1001	Jones	Shirley	2,345.82
A1002	Smythe	John	25,000.04
A1003	Smithe	Robert	31,000.04
A1004	Smithson	Ruth	21,000.04
A1005	Smiths	Peter	46,700.04
A1006	Smithers	Jack	33,275.04
A1007	Snell	George	26,780.04
A1008	Sneddon	Alison	450,000.04
A1009	Snashall	Damian	31,000.04
41010	Perry	Kate	60,000.04
A1011	Perron	Christopher	25,000.04
A1012	Paul	Patrick	26,456.04
1012	Dattincon	Goorgo	79 077 04

To see how the example works, copy this code and paste it to a form component:

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Height(346) Left(245) Top(13
Define_Com Class(#PRIM_LTVW) Name(#EMPLIST) Displayposition(1) Fu
Define_Com Class(#PRIM_LVCL) Name(#LVCL_1) Displayposition(1) Paren
Define_Com Class(#PRIM_LVCL) Name(#LVCL_2) Displayposition(2) Paren
Define_Com Class(#PRIM_LVCL) Name(#LVCL_3) Displayposition(3) Paren
Define_Com Class(#PRIM_LVCL) Name(#LVCL_4) Displayposition(4) Paren
Define_Com Class(#PRIM_LVCL) Name(#PB_RAISE) Caption('Give Raise')
Evtroutine Handling(#com_owner.Initialize)
Select Fields(#EMPLIST) From_File(PSLMST)
Add_Entry To_List(#EMPLIST)
Endselect
Endroutine
Evtroutine Handling(#PB_RAISE.Click)
```

Change Field(#SALARY) To('#SALARY \* 1.10') Upd\_Entry In\_List(#EMPLIST) Endroutine Evtroutine Handling(#EMPLIST.ItemGotSelection) Set Com(#PB\_RAISE) Enabled(TRUE) Endroutine End\_Com

## **Updating All Items in a List**

This example shows how you can work with all the items in a list using the SELECTLIST statement and then update the items using the UPD\_ENTRY command. The list view contains employee names and their salaries. Every time the "Give Raise" push button is clicked every employee in the list has their salary increased by 10%.

Employee Number	Employee Surname	Employee Given Name(s)	Employee Sal	
A0070	Brown	Veronica	60,651.25	
A0090	Bloggs	Fred	24,255.55	
A0193	Smithson	Fred	42,350.04	
A0907	Simpson	Anne	41,397.77	
A1001	Jones	Shirley	2,838.44	
A1002	Smythe	John	30,250.04	
A1003	Smithe	Robert	37,510.04	
A1004	Smithson	Ruth	25,410.04	
A1005	Smiths	Peter	56,507.04	
A1006	Smithers	Jack	30,250.04	
A1007	Snell	George	32,403.84	
A1008	Sneddon	Alison	544,500.04	
A1009	Snashall	Damian	37,510.04	
A1010	Perry	Kate	72,600.04	
A1011	Perron	Christopher	30,250.04	
A1012	Paul	Patrick	32,011.80	
1012	Dattincon	Goorgo	05 562 21	*

To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(334) LEFT(245) T
DEFINE_COM class(#PRIM_LTVW) name(#EMPLIST) DISPLAYPOSITIO
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_1) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_3) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_4) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#PB_RAISE) CAPTION('Give R
EVTROUTINE handling(#com_owner.Initialize)
SELECT fields(#EMPLIST) from_file(PSLMST)
ADD_ENTRY to_list(#EMPLIST)
ENDSELECT
```

ENDROUTINE EVTROUTINE handling(#PB\_RAISE.Click) SELECTLIST named(#EMPLIST) CHANGE field(#SALARY) to('#SALARY \* 1.10') UPD\_ENTRY in\_list(#EMPLIST) ENDSELECT ENDROUTINE END\_COM

## Deleting an Item from a List

Use the DLT\_ENTRY command to delete the selected item in a list.

This example fills a list view with the employee and their salaries. When the push button "Delete" is clicked the selected employee is deleted from the List View:

Employee Number	Employee Surname	Employee Given Name(s)	Employee Sal	
A0070	Brown	Veronica	50,125.00	
A0090	Bloggs	Fred	20,045.91	
A0193	Smithson	Fred	35,000.04	
A0907	Simpson	Anne	34,213.04	
A1001	Jones	Shirley	2,345.82	
A1002	Smythe	John	25,000.04	
A1003	Smithe	Robert	31,000.04	
A1004	Smithson	Ruth	21,000.04	
A1005	Smiths	Peter	46,700.04	
A1006	Smithers	Jack	25,000.04	
A1007	Snell	George	26,780.04	
A1008	Sneddon	Alison	450,000.04	
A1009	Snashall	Damian	31,000.04	
A1010	Perry	Kate	60,000.04	
A1011	Perron	Christopher	25,000.04	
A1012	Paul	Patrick	26,456.04	
1012	Dattincon	Goorgo	70 077 04	*

To see how the example works, copy this code and paste it to a form component:

#### FUNCTION options(\*DIRECT)

```
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(331) LEFT(245) T
DEFINE_COM class(#PRIM_LTVW) name(#EMPLIST) DISPLAYPOSITIO
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_1) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_3) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_4) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#PB_DELETE) CAPTION('Delet
EVTROUTINE handling(#com_owner.Initialize)
SET com(#com_owner) CAPTION(*component_desc)
SELECT fields(#EMPLIST) from_file(PSLMST)
ADD_ENTRY to_list(#EMPLIST)
ENDSELECT
```

ENDROUTINE EVTROUTINE handling(#PB\_DELETE.Click) DLT\_ENTRY from\_list(#EMPLIST) \* Following would be RDML to delete the record from the physical file \* and to do any file validations ENDROUTINE EVTROUTINE handling(#EMPLIST.ItemGotSelection) SET com(#PB\_DELETE) ENABLED(TRUE) ENDROUTINE ENDROUTINE END\_COM

## **Dynamically Assigning Icons in a List View**

Use the Image property of the CurrentItem in a list view to dynamically assign it an icon.

This example fills a list view with the employee details. An icon is displayed for the selected employee in the list view.

Employee Number	Employee Surname	Employee Given Name(s)	Employee Sal	
A0070	Brown	Veronica	50,125.00	
A0090	Bloggs	Fred	20,045.91	
A0193	Smithson	Fred	35,000.04	
A0907	Simpson	Anne	34,213.04	
A1001	Jones	Shirley	2,345.82	
A1002	Smythe	John	25,000.04	
A1003	Smithe	Robert	31,000.04	
🚡 A1004 📐	Smithson	Ruth	21,000.04	
A1005 13	Smiths	Peter	46,700.04	
A1006	Smithers	Jack	25,000.04	
A1007	Snell	George	26,780.04	
A1008	Sneddon	Alison	450,000.04	
A1009	Snashall	Damian	31,000.04	
A1010	Perry	Kate	60,000.04	Ŧ

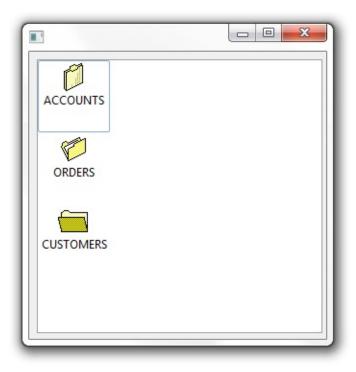
To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(346) LEFT(381) T
DEFINE_COM class(#PRIM_LTVW) name(#EMPLIST) DISPLAYPOSITIO!
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_1) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_3) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_4) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_4) DISPLAYPOSITION(
EVTROUTINE handling(#com_owner.Initialize)
SELECT fields(#EMPLIST) from_file(PSLMST)
ADD_ENTRY to_list(#EMPLIST)
ENDSELECT
ENDROUTINE
EVTROUTINE handling(#EMPLIST.ItemGotSelection) options(*NOCLEAR]
SET com(#EMPLIST.CURRENTITEM) IMAGE(#VI_EMPLOY)
```

#### ENDROUTINE EVTROUTINE handling(#EMPLIST.ItemLostSelection) options(\*NOCLEAR SET com(#EMPLIST.CURRENTITEM) IMAGE(\*NULL) ENDROUTINE END\_COM

### **Creating a Desktop Menu Using Large Icons in List View**

In this example a list view in LargeIcon view acts as a simple desktop style menu for "Accounts", "Orders" and "Customers". Double-clicking on the icon invokes the option. The ViewStyle property of the list view is set to Icon.



To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
```

BEGIN\_COM role(\*EXTENDS #PRIM\_FORM) HEIGHT(337) LEFT(381) T DEFINE\_COM class(#PRIM\_LTVW) name(#LTVW\_1) DISPLAYPOSITION DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_1) DISPLAYPOSITION( EVTROUTINE handling(#com\_owner.Initialize) INVOKE method(#com\_owner.uLargeIcon) U\_MENU(Accounts) U\_ICON(# INVOKE method(#com\_owner.uLargeIcon) U\_MENU(Orders) U\_ICON(#VI\_ INVOKE method(#com\_owner.uLargeIcon) U\_MENU(Orders) U\_ICON(#VI\_ INVOKE method(#com\_owner.uLargeIcon) U\_MENU(Customers) U\_ICON(# INVOKE method(#com\_owner.uLargeIcon) U\_MENU(Customers) U\_ICON(# ENDROUTINE MTHROUTINE name(uLargeIcon) DEFINE\_MAP for(\*INPUT) class(#std\_TEXTS) name(#u\_Menu) DEFINE\_MAP for(\*INPUT) class(#PRIM\_Icon) name(#u\_Icon) pass(\*by\_ref CHANGE field(#STD\_TEXTS) to('#U\_MENU.VALUE') ADD ENTRY to list(#LTVW 1) SET com(#LTVW\_1.CurrentItem) IMAGE(#u\_Icon) ENDROUTINE

EVTROUTINE handling(#ltvw\_1.doubleclick) options(\*NOCLEARMESSAC USE builtin(MESSAGE\_BOX\_SHOW) with\_args(OK OK INFORMATION " ENDROUTINE END\_COM

## **Sorting List Views 1**

List view columns can be sorted in ascending or descending order by setting the SortOnClick property of the columns to True.

This example fills a list view with employee number, name, salary and zip code details. The user can dynamically sort the list at any time by clicking on the heading of any column.

Employee Num	Employee Surname	Employee Given Name	Employee Sal	Post / Zip Code	
A1020	Douglas	Adam v3	121,500.04	2147	
A1027	Morrison	Alan	1,878,773.04	2007	
A1008	Sneddon	Alison	450,000.04	2160	
A1028	Maxwell	Andrew	75,500.00	2053	
A0907	Simpson	Anne	34,213.04	2145	
A1015	Woods	Bradley	313,000.04	2030	
A1019	Dickens	Charles	45,000.04	2147	
A1011	Perron	Christopher	25,000.04	2003	
A1009	Snashall	Damian	31,000.04	2150	
A1023	Reid	David	21,000.04	2001	
A1234	Jackson	Diane	72,125.00	2125	
A0193	Smithson	Fred	35,000.04	2034	
A0090	Bloggs	Fred	20,045.91	2220	
A3564	Brown	Freddy	30,000.00	2153	
A1017	Neave	Gary	25,600.04	2016	
A1013	Pattinson	George	78,977.04	2016	-
A1007	Cooll	Goorgo	26 790 04	2164	*

To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
```

```
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(330) LEFT(327) T
DEFINE_COM class(#PRIM_LTVW) name(#EMPLIST) DISPLAYPOSITIO
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_1) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_3) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_4) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_5) DISPLAYPOSITION(
EVTROUTINE handling(#com_owner.Initialize)
SELECT fields(#EMPLIST) from_file(PSLMST)
ADD_ENTRY to_list(#EMPLIST)
ENDSELECT
```

ENDROUTINE END\_COM

### **Sorting List Views 2**

This example extends Sorting List Views 1 by adding a second list view titled "SELECTLIST Employee Order" which is refilled every time a button titled "Show SELECTLIST Order" is clicked. It demonstrates that the order of the items in the list view remains constant (ie: in the order the items were added by the ADD\_ENTRY command) regardless of how the user has sorted visual representation of the list

Employee Num	Employee Surname	Employee Given Name	Employee Sal	Post / Zip Code	
A1018	Zacharia	Paul	25,900.04	2021	
A1015	Woods	Bradley	313,000.04	2030	
A1111	Verey	Warren	45,678.04	2345	
A1016	Turner	Jack	22,000.04	2100	
A1030	Turner	Valerie	45,000.04	2111	
A1022	Thompson	Kelly	25,000.29	2144	
A1024	Taylor	John	15,000.04	2002	
A1007	Snell	George	26,780.04	2164	
A1008	Sneddon	Alison	450,000.04	2160	
A1009	Snashall	Damian	31,000.04	2150	
A1002	Smythe	John	25,000.04	2100	
A0193	Smithson	Fred	35,000.04	2034	-
A1004	Smithcon	Duth	21 000 04	21.4.4	*
Employee Num	Employee Surname	Employee Given Name		Post / Zip Code	
A0070	Brown	Veronica	50,125.00	2153	*
A0090	Bloggs	Fred	20,045.91	2220	
A0193	Smithson	Fred	35,000.04	2034	
A0907	Simpson	Anne	34,213.04	2145	
A1001	Jones	Shirley	2,345.82	2001	
A1002	Smythe	John	25,000.04	2100	
A1003	Smithe	Robert	31,000.04	2000	
	Smithson	Ruth	21,000.04	2144	
	Smiths	Peter	46,700.04	2147	
A1004	Striftens	Jack	25,000.04	2050	
A1004 A1005	Smithers	Jack			
A1004 A1005 A1006		George	26,780.04	2164	
A1004 A1005 A1006 A1007 A1008	Smithers		26,780.04 450,000.04	2164 2160	

To see how the example works, copy this code and paste it to a form component:

FUNCTION options(\*DIRECT)

BEGIN COM role(\*EXTENDS #PRIM FORM) HEIGHT(594) LEFT(327) T DEFINE COM class(#PRIM LTVW) name(#EMPLIST) DISPLAYPOSITIO DEFINE COM class(#PRIM LVCL) name(#LVCL 1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_2) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_3) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_4) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_5) DISPLAYPOSITION( DEFINE COM class(#PRIM LTVW) name(#WRKLIST) DISPLAYPOSITIC DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_6) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_7) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_8) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_9) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_10) DISPLAYPOSITION DEFINE COM class(#PRIM PHBN) name(#PB ORDER) CAPTION('Show DEFINE COM class(#PRIM LABL) name(#LABL 1) CAPTION('SELECTI EVTROUTINE handling(#com owner.Initialize) SELECT fields(#EMPLIST) from file(PSLMST) ADD ENTRY to list(#EMPLIST) **ENDSELECT ENDROUTINE** EVTROUTINE handling(#PB\_ORDER.Click) CLR\_LIST named(#WRKLIST) SELECTLIST named(#EMPLIST) ADD\_ENTRY to\_list(#WRKLIST) **ENDSELECT ENDROUTINE** END COM

## Saving and Restoring a List from a File

This example shows how you can save and load the contents of a list to a file. It displays a List View containing employee details.

When the Load from DBMS button is clicked the employee list is loaded from the PSLMST table. When the Load from File button is clicked, the employee list is loaded from a file named c:\Employee\_List.dat which is in tab delimited format. When the Save to File button is clicked, the employee list is saved into a file named c:\Employee\_List.dat in tab delimited format.

Employee Num	Employee Surn	Employee Given Nam	Employee Salary
A0070	Brown	Veronica	50,125.00
A0090	Bloggs	Fred	20,045.91
A0193	Smithson	Fred	35,000.04
A0907	Simpson	Anne	34,213.04
A1001	Jones	Shirley	2,345.82
A1002	Smythe	John	25,000.04
A1003	Smithe	Robert	31,000.04
A1004	Smithson	Ruth	21,000.04
A1005	Smiths	Peter	46,700.04
A1006	Smithers	Jack	25,000.04
A1007	Snell	George	26,780.04
A1008	Sneddon	Alison	450,000.04
A1009	Snashall	Damian	31,000.04
A1010	Perry	Kate	60,000.04
A1011	Perron	Christopher	25,000.04
A1012	Paul	Patrick	26,456.04
1012	Dattincon	Goorgo	79 077 04
Load from DBMS	Load from F	Save to File	Clear List

To see how the example works, copy this code and paste it to a form component:

```
FUNCTION options(*DIRECT)
BEGIN_COM role(*EXTENDS #PRIM_FORM) HEIGHT(346) LEFT(327) T
DEFINE_COM class(#PRIM_LTVW) name(#EMPLIST) DISPLAYPOSITIO
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_1) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_2) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_3) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#LVCL_4) DISPLAYPOSITION(
DEFINE_COM class(#PRIM_LVCL) name(#PB_DBMS) CAPTION('Load fr
DEFINE_COM class(#PRIM_PHBN) name(#PB_FILE) CAPTION('Load from the set of the set
```

DEFINE COM class(#PRIM PHBN) name(#PB SAVE) CAPTION('Save to 1 DEFINE COM class(#PRIM PHBN) name(#PB CLEAR) CAPTION('Clear I DEFINE field(#RETCODE) type(\*CHAR) length(2) EVTROUTINE handling(#com\_owner.Initialize) **ENDROUTINE** EVTROUTINE handling(#PB\_DBMS.Click) CLR\_LIST named(#EMPLIST) SELECT fields(#EMPLIST) from\_file(PSLMST) ADD\_ENTRY to\_list(#EMPLIST) **ENDSELECT ENDROUTINE** EVTROUTINE handling(#PB\_FILE.Click) CLR\_LIST named(#EMPLIST) USE builtin(TRANSFORM\_FILE) with\_args(#EMPLIST 'C:\EMPLOYEE\_L **ENDROUTINE** EVTROUTINE handling(#PB\_SAVE.Click) USE builtin(TRANSFORM\_LIST) with\_args(#EMPLIST 'C:\EMPLOYEE\_L] **ENDROUTINE** EVTROUTINE handling(#PB\_CLEAR.Click) CLR\_LIST named(#EMPLIST) **ENDROUTINE** END\_COM

# **Special Lists: Graph**

Graphs show one or more field's information in a graphical format. In the current version you can display area, bar, line, pie, scatter and surface charts. For more information refer to the Features online help of the GraphType property.

A pie chart displays data from a single field whereas the other charts can display data from several fields. Scatter and surface graphs require a minimum of two data fields.

As in other lists, the fields are represented as columns. The columns that contain data to be graphed (numeric data) are defined as ColumnRole(Data) the columns which contain textual information are defined as ColumnRole(Label). The ColumnRole can also be set to None which hides and disables the column. This is useful when you have several columns in your graph and you change the data being displayed dynamically.

The different graph types can have different style settings. These are set using the AreaStyle, BarStyle, LineStyle, PieStyle, ScatterStyle and SurfaceStyle properties.

When you are graphing minimum, maximum, average and total values, you can use the KEEP\_XXXXX commands to calculate the values. For instance you could display average salaries per section by defining a #SALARYAVG field and storing the average value of the #SALARY field in it:

```
*(Department loaded and selected previously)
EVTROUTINE HANDLING(#PHBN_1.Click)
clr_list #grph_1
select (#section #secdesc) from_file(sectab) with_key(#deptment)
select #salary from_file(pslmst1) with_key(#deptment #section)
KEEP_AVG OF_FIELD(#SALARY) IN_FIELD(#SALARYAVG)
endselect
add_entry #grph_1
endselect
ENDROUTINE
```

Graphs receive the ItemGotSelection event when the user clicks on a section of the graph. This event will set the current entry for the graph columns. For instance if you click on bar representing an employee in a bar chart, that employee becomes the current entry in the graph. You can use this feature to provide more information about the current entry for instance in a message box or another graph.

The following pie chart shows the percentage of total salaries earned by individual employees.



If you run the graph, notice how the surname (CurrentItem) changes when you click on different sections of the pie (ItemGotSelection event).

Here is the source for the graph:

FUNCTION OPTIONS(\*DIRECT)

```
*Definitions
BEGIN_COM HEIGHT(412) LEFT(218) TOP(160) WIDTH(462)
DEFINE_COM CLASS(#PRIM_GRPH) NAME(#GRAPH) DISPLAYPOSIT
DEFINE_COM CLASS(#PRIM_GRCL) NAME(#GRCL_1) DISPLAYPOSIT
DEFINE_COM CLASS(#PRIM_GRCL) NAME(#GRCL_2) COLUMNROLE
DEFINE_COM CLASS(#SURNAME.Visual) NAME(#SURNAME) DISPLA
```

\*Get data, add it to the graph and set the caption and the type EVTROUTINE HANDLING(#graph.initialize)

CLR\_LIST NAMED(#GRAPH) SELECT FIELDS(#SALARY #SURNAME) FROM\_FILE(PSLMST) WHE ADD\_ENTRY TO\_LIST(#GRAPH) ENDSELECT Set #GRAPH CAPTION('Salaries by Employee' ) Set #graph GRAPHTYPE(Pie) ENDROUTINE

END\_COM

### Special Lists: Multi-line Edit Box



A multi-line edit box appears to the user as an edit box with multiple lines. To the developer it is a list in which every line is a separate entry. To fill an edit box from a file use the ADD\_ENTRY command and to select data from an edit box to be written to a file use the SELECTLIST command.

A multi-line edit box has to contain at least one column sourced from a field which is used to store the data. Set the ColumnRole property of this column to Data. Optionally a column can be used to store the line number (ColumnRole LineNumber). If you are using the AddEntryProperty MultiplePerRow you would usually also have a column for the row number (ColumnRole LineContinuation).

When the data is fetched from a file to the edit box or extracted from the edit box, the AddEntryMode of the edit box is used to specify whether to process one entry per line (OnePerLine) or whether to process multiple entries per line (MultiplePerLine).

The length of the line is set using the MaximumLineLength property. With AddEntryMode OnePerLine the line cannot be longer than the field from which the data column is sourced.

When a file is updated from the multi-line edit box and the AddEntryMode is MultiplePerLine, the SELECTLIST command gets the data from the edit box in chunks equal to the length of the source field and the original line number is recorded in the field with the ColumnRole of LineNumber and the row number is recorded in the field with the ColumnRole LineContinuation. When data is written from the file to the edit box, the entries are packed together to fill the line. The multi-line edit box has a Modified property. This property is set to True when any changes are made to the contents of the box. The value of this property is not reset automatically, so in your application you need to set it to False after you have dealt with the changes.

A multi-line edit box has methods which you often need in an editor such as cut, copy and paste, print, undo and redo. It also has properties which you can use to, for instance, determine the current line and the position of the cursor on it, to display a vertical or horizontal scroll bar and to control whether line numbers are shown.

The following code shows how you carry out the basic multi-line edit box operations. The examples use a NoteFile file which consists of these fields:

Field	Description	Туре	Length and Decimals	Key
NoteNo	Number identifying this particular note	Packed	7,0	1
LineNo	Line number	Packed	7,0	2
RowNo	Row number	Packed	7,0	3
Text	The text	Alpha	50	

The name of the multi-line edit box is #MeditBox. It is set up like this:

Property	Value
AddEntryMode of #MeditBox	MultiplePerLine
MaximumLineLength of #MeditBox	1000
ShowLineNumbers	True
WordWrap of #MeditBox	True

#MeditBox has these columns:

Column	Source Field	ColumnRole Property
Column 1	Text	Data
Column 2	LineNo	LineNumber

Column 3 RowNo LineContinuation

	· · · · · · · · · · · · · · · · · · ·	
Note number	1234567	
	20001 ABCDEFGHIJKLMNOPQRSTUVWXY ZABCDEFGHIJKLMNOPQRSTUVWX	Select All
Update		Cut
Get Cet		Сору
		Paste
Clear	· · · · · · · ·	Undo
Delete		Redo
Delete	· · · · · · · · · · · · · · · · · · ·	Find
		Replace
		Print

The Note Number field from the NoteFile is used to identify the note.

On the left, the form has buttons to:

- Fill the edit box from the file
- Update the note file
- Clear the edit box
- Delete an entry in the file.

The buttons have these Click event routines:

\* This code deletes any previous text and writes the contents of the

\* edit box to the file: EVTROUTINE HANDLING(#UPDATEB.Click) if '#meditbox.modified \*eq True'

II #Ineditbox.inodified "eq True

delete from\_file(notefile) with\_key(#noteno)

selectlist #meditbox

insert fields(#noteno #lineno #rowno #text) to\_file(notefile) val\_error(\*next)
endselect

endif

ENDROUTINE

\*This fetches the text lines of a note from the file: EVTROUTINE HANDLING(#GETBTN.Click) select fields(#text #lineno #rowno) from\_file(notefile) with\_key(#noteno) add\_entry (#MEditBox)

#### ENDSELECT ENDROUTINE

\*This clears the edit box: EVTROUTINE HANDLING(#CLEARBTN.Click) clr\_list (#meditbox) ENDROUTINE

\*This deletes the entry from the database and clears the edit box: EVTROUTINE HANDLING(#DELETEBTN.Click) dowhile '#io\$sts \*ne NR' delete from\_file(notefile) with\_key(#noteno) endwhile clr\_list (#meditbox) ENDROUTINE

On the right there are several buttons used to invoke the methods of a multi-line edit box. They contain these event routines:

EVTROUTINE HANDLING(#SELECTALL.Click) invoke #meditbox.selectall ENDROUTINE

EVTROUTINE HANDLING(#CUT.Click) invoke #meditbox.cut ENDROUTINE

EVTROUTINE HANDLING(#COPY.Click) invoke #meditbox.copy ENDROUTINE

EVTROUTINE HANDLING(#PASTE.Click) invoke #meditbox.paste ENDROUTINE

EVTROUTINE HANDLING(#UNDO.Click) invoke #meditbox.undo ENDROUTINE EVTROUTINE HANDLING(#REDO.Click) invoke #meditbox.redo ENDROUTINE

EVTROUTINE HANDLING(#FIND.Click) invoke #meditbox.find ENDROUTINE

EVTROUTINE HANDLING(#REPLACE.Click) invoke #meditbox.replace ENDROUTINE

EVTROUTINE HANDLING(#PRINT.Click) invoke #meditbox.print ENDROUTINE

↑ 6.15.8 Working with Items in Lists

# **Error Handling in a Grid/List**

The following example shows how you can use keyed collections to perform error checking for grids where the user may update multiple records at once. It also demonstrates the correct use of nested BEGINCHECK/ENDCHECK blocks to provide visual feedback for fields which fail validation rules.

Empno	Surname	Zip	Employee Salary		<u>R</u> efresh
A0070	Brown	2153	50,125.00		
A0090	Bloggs	2220	20,045.91		Update
A0193	Smithson	2034	35,000.04		
A0907	Simpson	2145	34,213.04		
A1001	Jones	2001	2,345.82		
A1002	Smythe	2100	25,000.04		
A1003	Smithe	2000	31,000.04		
A1004	Smithson	2144	21,000.04		
A1005	Smiths	2147	46,700.04		
A1006	Smithers	2050	25,000.04	-	

Note that nested BEGINCHECK/ENDCHECK blocks are used because updating a grid in this way is similar to validating multiple details input via a browse list on IBM i.

\* COMPONENT: STD\_FORM

Function Options(\*DIRECT)

```
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(221) Clientwidth
Define_Com Class(#PRIM_GRID) Name(#GRID_1) Captionnoblanklines(Tru
Define_Com Class(#PRIM_GDCL) Name(#GDCL_1) Caption('Empno') Capti
Define_Com Class(#PRIM_GDCL) Name(#GDCL_2) Caption('Surname') Caption
Define_Com Class(#PRIM_GDCL) Name(#GDCL_3) Caption('Zip') Captiont
Define_Com Class(#PRIM_GDCL) Name(#GDCL_4) Columnalign(Right) Di
Define_Com Class(#PRIM_GDCL) Name(#GDCL_4) Columnalign(Right) Di
Define_Com Class(#PRIM_FHBN) Name(#PHBN_1) Caption('&Refresh ') Di
Define_Com Class(#PRIM_STBR) Name(#STBR_1) Displayposition(3) Heig
Define_Com Class(#PRIM_PHBN) Name(#PHBN_upd) Caption('&Update ')
*
```

Define\_Com Class(#prim\_kcol<#postcode #empno>) Name(#postcodes)
Define\_Com Class(#prim\_kcol<#salary #empno>) Name(#salaries)
\*

Evtroutine Handling(#PHBN\_1.Click)

\* Clear Collections as well as grid

```
#postcodes.RemoveAll
#salaries.RemoveAll
Clr List Named(#grid 1)
Select Fields(#grid 1) From File(pslmst)
#postcodes<#empno> := #postcode
#salaries<#empno> := #salary
Add_Entry To_List(#grid_1)
Endselect
Endroutine
Evtroutine Handling(#PHBN_upd.Click)
* Check the input data
Begincheck
Selectlist Named(#GRID 1)
Continue If(#Com_owner.HasNotChanged)
Begincheck
Update Fields(#salary #postcode) In_File(pslmst) With_Key(#empno) Io_Erro
If_Status Is_Not(*okay)
#grid_1.currentitem.Visualstyle <= #vs_emph</pre>
Else
#grid_1.currentitem.Visualstyle <= #vs_norm</pre>
Endif
Upd_Entry In_List(#GRID_1)
Endcheck If_Error(*next)
Endselect
Endcheck
* Now do the updates (now that every field is validated and correct)
Selectlist Named(#Grid_1)
Continue If(#Com_owner.HasNotChanged)
Update Fields(#salary #postcode) In File(pslmst) With Key(#empno) Io Erro
Endselect
Endroutine
* Method Routine to check for changes
Mthroutine Name(HasNotChanged)
Define Map For(*result) Class(#prim boln) Name(#Result)
If ((#postcode = #postcodes<#Empno>) And (#salary = #salaries<#Empno>))
#Result := True
```

Endif Endroutine End\_Com

The "inner" BEGINCHECK/ENDCHECK loop is processed for each grid row entry that the user updated. The IF\_ERROR(\*NEXT) parameter causes the SELECTLIST loop to continue to process all grid entries and not stop the first time an error is detected.

If a validation error is detected, the "outer" BEGINCHECK/ENDCHECK command uses the default IF\_ERROR(\*LASTDIS) setting which will cause the update event routine to be terminated without writing any records to the database.

A validation error will be detected if an error is found in *one or more* of the grid rows. This happens because any error in the "inner" validation block also triggers an error in the "outer" validation block.

↑ 6.15.8 Working with Items in Lists

# 6.16 Tab Folder Framework

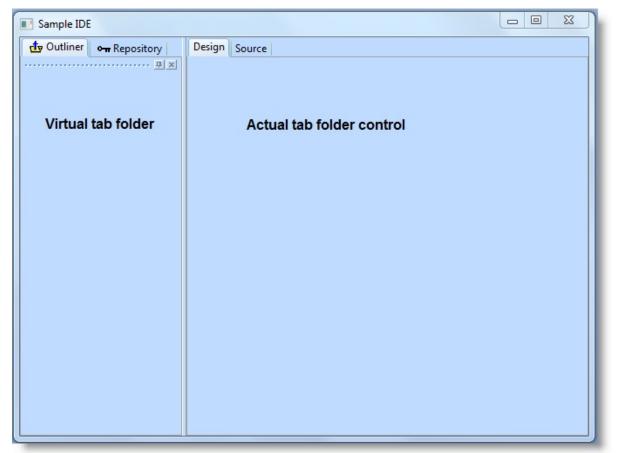
The tab folder control has advanced docking, undocking and autohiding features you can use to transform a single tab folder into a tab framework application with virtual tab folders attached to various parts of the window.

The virtual tab folders can be automatically hidden (autohide) and tab sheets can be moved and attached to other parts of the screen (undock and dock).

When you want to use these advanced features, the tab folder has to be centered by an attachment layout manager on a form. The structure of the tab framework is created by setting the DockPosition of individual tab sheets.

The tab folder properties manage the appearance of the framework and the individual tab sheets manage the position and layout of a page in the tab framework.

Here is an example of a basic Tab Framework Application. It consists of a single tab folder attached to the form using an Attachment layout manager. The Outliner and Repository tab sheets are attached to the left using their DockPosition property. This causes a virtual tab folder to appear on the left.



The virtual tab folder is created by setting the DockPosition property of two tab sheets to Left.

The actual tab folder control is attached to the centre of the form using an attachment.

You can create this example by copying and pasting the 6.16.7 Tab Folder Example Source into a form.

	The LANSA Editor has been built u	using the features described here.
6.	16.1 Virtual Tab Folders	6.16.5 Autohiding Virtual Tab Folders
	16.2 Undocking and Docking Tab	6.16.6 Tab Framework Events and Methods
	16.3 Closing a Tab Sheet	6.16.7 Tab Folder Example Source
6.	16.4 Virtual Tab Folder Properties	
$\uparrow$		

귀

1 6. Creating Applications Using Components

# 6.16.1 Virtual Tab Folders

You can create virtual tab folders to the left, right, top and bottom of the window using the Left, Right, Top or Bottom DockPosition property values of individual tab sheets:

<b>Docking Fram</b> Page3	ework Sample	
<u>₽</u>	С	
Page1	Page5	Page2
A	E	В
Page4	D	

On picture Tab sheet Caption DockPosition

А	TBSH_1	Page 1	Left
В	TBSH_2	Page 2	Right
С	TBSH_3	Page 3	Тор
D	TBSH_4	Page 4	Bottom
E	TBSH_5	Page 5	Center (default)
<b>`</b>			

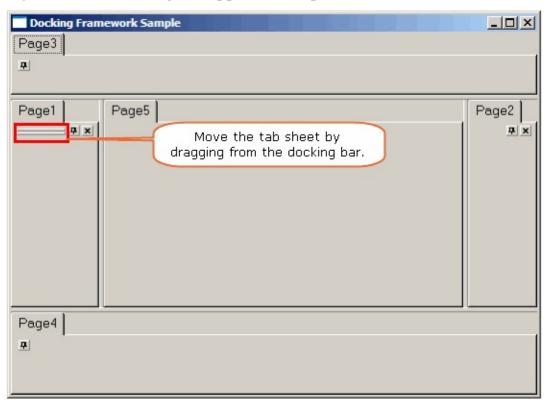
1 6.16 Tab Folder Framework

# 6.16.2 Undocking and Docking Tab Sheets

The tab sheets in the virtual tab folders can be moved and attached to other parts of the screen (undocked and docked) using the DockAllowUndock and DockAllowedPositions properties of the tab sheet.

Docking and Undocking properties are specified for every individual tab sheet.

In this example the DockAllowUndock has been set to True for tab sheet Page 1 so can be undocked from the left tab folder. When the DockAllowUndock property is True, a docking bar appears on top of the tab sheet:



Use the DockAllowedPositions property to specify which part of the screen the tab sheet can be attached to (docked). The possible values are:

- Left
- Right
- Top
- Bottom
- None

All positions can be selected. None overrides the multi-select options and specifies that the tab sheet cannot be docked anywhere.

For example, if DockAllowedPositions for tab sheet Page 1 is Left and Bottom, it can only be moved and attached to the virtual tab folder on the bottom of the window (and back to its original position):

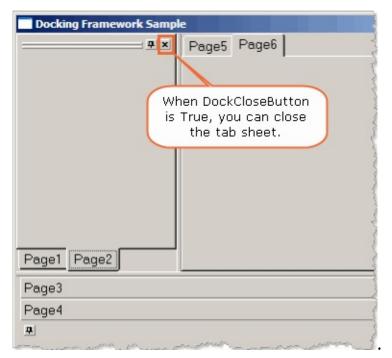
Page3	ng Framework Sample	4-
Page5		P
Page1 ≭ ₹	Page4 Tab sheet Page 1 has been undocked from the left and docked to the bottom of the window.	

Note that this property applies to user interaction only and has no bearing on programmatic changes. Therefore a tab sheet with DockPosition(Left) and DockAllowedPositions(None) will appear as a DockLeft tab sheet.

1 6.16 Tab Folder Framework

# 6.16.3 Closing a Tab Sheet

If you want to allow the user to close the tab sheet, use its DockCloseButton property to specify whether a Close button is displayed in the tab sheet. This property can be set to True or False:



When a tab sheet has been closed, its Hidden property is set to True.

Use this property with the ShowSheet and HideSheet methods to determine displaying or redisplaying of a tab sheet.

Note that the Hidden property will be True if a tab sheet has been undocked and the User closes the undock form.

1 6.16 Tab Folder Framework

# **6.16.4 Virtual Tab Folder Properties**

The four virtual tab folders (left, right, top and bottom) have individual properties that control their appearance and behavior. In other words, the virtual tab folders can be different from each other and the actual tab folder control.

In this example the tab framework has a left and bottom virtual tab folder in addition to the actual tab folder control:

Docking Framework Samp	e	
	Page5 Page6	
<b>A</b>	В	
Page1 Page2		 
Page3		
Page4		
₽ C		

These are the relevant properties of the tab folder:

<b>A</b> Left virtual tab folder	LeftTabLocation	Bottom	The tabs in the virtual left tab folder are placed on the bottom of the folder
	LeftTabWidth	167	The virtual left tab folder is 167 pixels wide.
	LeftViewStyle	Tabs	The tabs in the virtual left tab folder are tabs (as opposed to stacked).

<b>B</b> Actual tab folder control	TabLocation	Тор	The tabs in the actual tab folder are placed on top of the folder.
	Width	549	The tab folder is 549 pixels wide.
	ViewStyle	Tabs	The tabs in the actual tab folder are tabs (as opposed to stacked).
<b>C</b> Bottom virtual tab	BottomTabLocation	Тор	The tabs in the virtual bottom tab folder are placed on top of the folder.
folder	BottomTabHeight	125	The virtual bottom tab folder is 125 pixels high.
	BottomViewStyle	Stacked	The tabs in the virtual bottom tab folder are displayed stacked.
	an Ename ar saula		

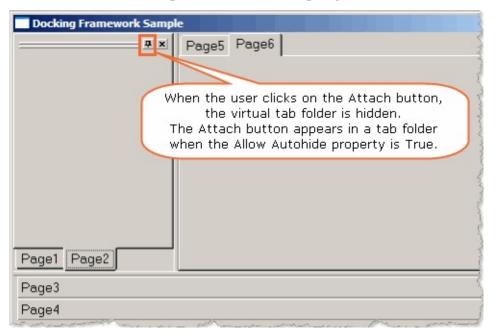
↑ 6.16 Tab Folder Framework

# 6.16.5 Autohiding Virtual Tab Folders

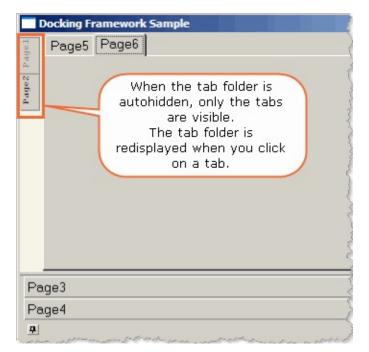
The virtual tab folders have properties that you can use to allow the user to automatically hide them (autohide).

Use Autohide to maximize the use of screen real estate.

When using autohide, the tab folder is automatically hidden when the user clicks on the Attach button (a pin icon) in the top right corner of a tab sheet:



When the tab folder is hidden, only the tabs of the tab sheets are shown. The tab folder appears again when the user clicks on a tab.



To end autohiding, the user clicks again on the Attach button.



The AutoHide properties are specific to a virtual tab folder (bottom, left, right or top virtual tab folder):

BottomAllowAutoHide	Use the AllowAutoHide properties to specify
LeftAllowAutoHide	whether or not a virtual tab folder can be
RightAllowAutoHide	automatically hidden.

TopAllowAutoHide	When this property is true, the Attach button is displayed in the top right corner of the tab sheets in the folder.
BottomAutoHide LeftAutoHide RightAutoHide TopAutoHide	Use the Autohide properties of virtual tab folders to programmatically autohide the tab folder or to find out whether the tab folder has been autohidden. The value of this property can be True or False. Note that this property has no effect at design time to make it easier to design an autohide tab folder.

↑ 6.16 Tab Folder Framework

# **6.16.6 Tab Framework Events and Methods**

A tab sheet has the following events to aid in the management of a tab framework application:

CloseClick	The CloseClick event is triggered when the user clicks on the DockCloseButton cross.
AutoHideClick	The AutoHideClick event is triggered when the user clicks on the AutoHideButton cross.
Docked	The Docked event is triggered when the user docks a tab sheet.
	This event is triggered when the user docks from an undocked position or when the dock position changes whilst staying docked (for example DockPosition changes from Left to Right)
Undocked	The Undocked event is triggered when the user undocks a tab sheet.

A tab sheet has the following methods to aid in the management of a tab framework application:

ShowSheet	The ShowSheet method can be used to redisplay a tab sheet that has been hidden via the click of the Close button.
HideSheet	The HideSheet method can be used to DockHide a tab Sheet. This is the programmatic equivalent of clicking the Close button.

### ↑ 6.16 Tab Folder Framework

### 6.16.7 Tab Folder Example Source

Following is the source for the form: Function Options(\*DIRECT) BEGIN COM ROLE(\*EXTENDS #PRIM FORM) CAPTION('Sample IDE') CLIENTHEIGHT(457) CLIENTWIDTH(652) HEIGHT(484) LAYOUTMANAGER(#ATLM\_1) LEFT(330) TOP(126) WIDTH(660) DEFINE COM CLASS(#PRIM TAB) NAME(#TAB 1) DISPLAYPOSITION(1) DRAGSTYLE(Automatic) DRAGTABS(True) HEIGHT(457) LEFT(0) LEFTTABWIDTH(186) PARENT(#COM OWNER) TABPOSITION(1) TABSTOP(False) TOP(0) WIDTH(652) DEFINE COM CLASS(#PRIM TBSH) NAME(#TBSH 1) CAPTION('Outliner') DISPLAYPOSITION(1) DOCKALLOWEDPOSITIONS(Left+Bottom+Right) DOCKALLOWUNDOCK(True) DOCKCLOSEBUTTON(True) DOCKPOSITION(Left) HEIGHT(430) IMAGE(#STD BTMAP) LEFT(4) OPENED(True) PARENT(#TAB\_1) TABPOSITION(1) TABSTOP(False) TOP(23) WIDTH(178) DEFINE\_COM CLASS(#PRIM\_TBSH) NAME(#TBSH\_2) CAPTION('Repository') DISPLAYPOSITION(2) DOCKALLOWEDPOSITIONS(Left+Bottom+Right) DOCKALLOWUNDOCK(True) DOCKCLOSEBUTTON(True) DOCKPOSITION(Left) HEIGHT(430) IMAGE(#VB LOCK) LAYOUTMANAGER(#ATLM 1) LEFT(4) PARENT(#TAB 1) TABPOSITION(2) TABSTOP(False) TOP(23) WIDTH(178) DEFINE\_COM CLASS(#PRIM\_TBSH) NAME(#TBSH\_4) CAPTION('Design') DISPLAYPOSITION(1) HEIGHT(431) LEFT(4) OPENED(True) PARENT(#TAB\_1) TABPOSITION(1) TABSTOP(False) TOP(22) WIDTH(453) DEFINE\_COM CLASS(#PRIM\_TBSH) NAME(#TBSH\_3) CAPTION('Source') DISPLAYPOSITION(2) HEIGHT(431) LAYOUTMANAGER(#ATLM\_1) LEFT(4) PARENT(#TAB\_1) TABPOSITION(2) TABSTOP(False) TOP(22) WIDTH(453) DEFINE\_COM CLASS(#PRIM\_TAB) NAME(#TAB\_2) DISPLAYPOSITION(1) HEIGHT(415) LEFT(0) PARENT(#TBSH\_2) TABLOCATION(Bottom) TABPOSITION(1) TABSTOP(False) TOP(15) WIDTH(178) DEFINE COM CLASS(#PRIM ATLM) NAME(#ATLM 1)

DEFINE\_COM CLASS(#PRIM\_ATLI) NAME(#ATLI\_1) ATTACHMENT(Center) MANAGE(#TAB\_2) PARENT(#ATLM\_1) DEFINE\_COM CLASS(#PRIM\_TBSH) NAME(#TBSH\_5) CAPTION('Fields') DISPLAYPOSITION(1) HEIGHT(389) LAYOUTMANAGER(#ATLM\_1) LEFT(4) OPENED(True) PARENT(#TAB\_2) TABPOSITION(1) TABSTOP(False) TOP(4) WIDTH(170)

DEFINE COM CLASS(#PRIM TBSH) NAME(#TBSH 6) CAPTION('Files') DISPLAYPOSITION(2) HEIGHT(389) LAYOUTMANAGER(#ATLM\_1) LEFT(4) PARENT(#TAB\_2) TABPOSITION(3) TABSTOP(False) TOP(4) WIDTH(170) DEFINE\_COM CLASS(#PRIM\_TBSH) NAME(#TBSH\_7) CAPTION('Forms') DISPLAYPOSITION(3) HEIGHT(389) LAYOUTMANAGER(#ATLM 1) LEFT(4) PARENT(#TAB 2) TABPOSITION(2) TABSTOP(False) TOP(4) WIDTH(170) DEFINE\_COM CLASS(#PRIM\_LTVW) NAME(#FIELDS) COMPONENTVERSION(2) DISPLAYPOSITION(1) FULLROWSELECT(True) HEIGHT(389) LEFT(0) PARENT(#TBSH\_5) SHOWSORTARROW(True) TABPOSITION(1) TOP(0) WIDTH(170) DEFINE\_COM CLASS(#PRIM\_ATLI) NAME(#ATLI\_2) ATTACHMENT(Center) MANAGE(#FIELDS) PARENT(#ATLM\_1) DEFINE\_COM CLASS(#PRIM\_LTVW) NAME(#FILES) COMPONENTVERSION(2) DISPLAYPOSITION(1) FULLROWSELECT(True) HEIGHT(389) LEFT(0) PARENT(#TBSH\_6) SHOWSORTARROW(True) TABPOSITION(1) TOP(0) WIDTH(170) DEFINE COM CLASS(#PRIM ATLI) NAME(#ATLI 3) ATTACHMENT(Center) MANAGE(#FILES) PARENT(#ATLM 1) DEFINE COM CLASS(#PRIM LTVW) NAME(#FORMS) COMPONENTVERSION(2) DISPLAYPOSITION(1) FULLROWSELECT(True) HEIGHT(389) LEFT(0) PARENT(#TBSH 7) SHOWSORTARROW(True) TABPOSITION(1) TOP(0) WIDTH(170) DEFINE COM CLASS(#PRIM ATLI) NAME(#ATLI 4) ATTACHMENT(Center) MANAGE(#FORMS) PARENT(#ATLM 1) DEFINE COM CLASS(#PRIM LVCL) NAME(#LVCL 1) CAPTION('Form Name') CAPTIONTYPE(Caption) DISPLAYPOSITION(1) PARENT(#FORMS) SOURCE(#STD TEXT) WIDTH(99) DEFINE COM CLASS(#PRIM LVCL) NAME(#LVCL 2) CAPTION('File Name') CAPTIONTYPE(Caption) DISPLAYPOSITION(1)

PARENT(#FILES) SOURCE(#STD TEXT) WIDTH(100) DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_3) CAPTION('Field Name') CAPTIONTYPE(Caption) DISPLAYPOSITION(1) PARENT(#FIELDS) SOURCE(#STD TEXT) WIDTH(95) DEFINE\_COM CLASS(#prim\_memo) NAME(#source) CURRENTLINE(1) DISPLAYPOSITION(1) HEIGHT(431) LEFT(0) MAXIMUMLINELENGTH(50) PARENT(#TBSH\_3) TABPOSITION(1) TOP(0) WIDTH(453) DEFINE\_COM CLASS(#PRIM\_ATLI) NAME(#ATLI\_5) ATTACHMENT(Center) MANAGE(#source) PARENT(#ATLM\_1) DEFINE\_COM CLASS(#PRIM\_MECL) NAME(#MECL\_1) COLUMNROLE(Data) DISPLAYPOSITION(1) PARENT(#source) SOURCE(#STD\_TEXT) DEFINE COM CLASS(#PRIM ATLI) NAME(#ATLI 6) ATTACHMENT(Center) MANAGE(#TAB 1) PARENT(#ATLM 1) Evtroutine Handling(#com\_owner.Initialize) #std text := 'ADDRESS1' Add\_Entry To\_List(#FIELDS) Set Com(#fields.CurrentItem) Image(#VI\_FOLDOP) #std\_text := 'SALARY' Add\_Entry To\_List(#FIELDS) Set Com(#fields.CurrentItem) Image(#VI\_FOLDOP) #std text := 'STD NUM' Add\_Entry To\_List(#FIELDS) Set Com(#fields.CurrentItem) Image(#VI\_FOLDOP) #std text := 'STD TEXT' Add Entry To List(#FIELDS) Set Com(#fields.CurrentItem) Image(#VI FOLDOP) #std text := 'PSMLST' Add Entry To List(#files) Set Com(#files.CurrentItem) Image(#VI EMPLOY) #std text := 'DEPTMENT' Add Entry To List(#files) Set Com(#files.CurrentItem) Image(#VI EMPLOY) #std text := 'PSLSKL' Add Entry To List(#files) Set Com(#files.CurrentItem) Image(#VI EMPLOY) #std text := 'FORM1' Add Entry To List(#forms)

```
Set Com(#forms.CurrentItem) Image(#STD_ICON)
#std text := 'FORM2'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std text := 'FORM3'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD_ICON)
#std text := 'FORM4'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD_ICON)
#std text := 'FORM5'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std text := 'FORM6'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std text := 'FORM7'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD_ICON)
#std text := 'FORM8'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD_ICON)
#std text := 'FORM9'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD_ICON)
#std text := 'FORM10'
Add Entry To List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std text := 'FORM11'
Add Entry To List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std text := 'FORM12'
Add Entry To List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std text := 'FORM13'
Add Entry To List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std text := 'FORM14'
Add Entry To List(#forms)
```

```
Set Com(#forms.CurrentItem) Image(#STD_ICON)
#std text := 'FORM15'
Add_Entry To_List(#forms)
Set Com(#forms.CurrentItem) Image(#STD ICON)
#std_text := 'Function Options(*DIRECT)'
Add_Entry To_List(#source)
#std_text := ' Begin_Com Role(*EXTENDS #PRIM_FORM)
Clientheight(541) Clientwidth(825) Height(575) Left(340) Top(178)
Width(833)'
Add_Entry To_List(#source)
               Define_Com Class(#PRIM_TAB) Name(#TAB_1)
#std text := '
Displayposition(1) Docklefttabwidth(157) Height(489) Left(48)
Parent(#COM_OWNER) Tabposition(1) Tabstop(False) Top(16) Width(713)
\n End Com'
Add_Entry To_List(#source)
#std text := 'End Com'
Add_Entry To_List(#source)
Endroutine
End Com
```

1 6.16 Tab Folder Framework

### 6.17 Menus

Object	Edit	1
	Undo Redo	
	Options +	User
L		Network
		V5

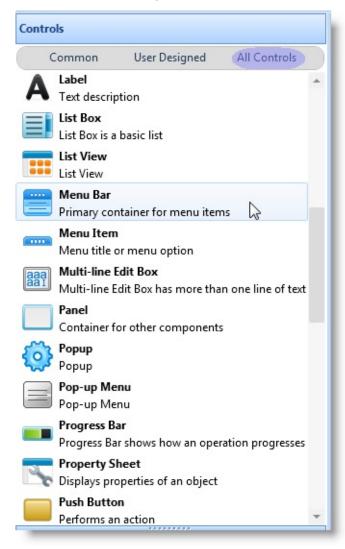
A menu consists of a menu bar, submenus and menu items.

A menu item can be either a first-level item (menu title) such as Object and Edit in the above example, or a subordinate item (menu option) as are all the other items in the example. A submenu can belong to a menu title or to a menu option--for example the menu option Preferences has its own submenu.

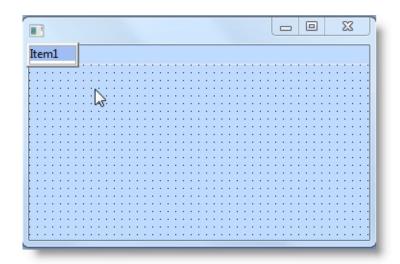
- 6.17.1 Creating Menus
- 6.17.2 Accelerator Keys
- 6.17.3 Shortcut Keys
- 6.17.4 A Few Things to Remember about Keyboard Access
- 6.17.5 Writing the Menu Events
- 6.17.6 Creating a Standard Menu Automatically
- 6.17.7 Creating a Popup-Menu
- ↑ 6. Creating Applications Using Components

## 6.17.1 Creating Menus

You need a menu bar to contain the menu items. So, to create a menu, first drag a Menu bar from the Controls tab to your form.



The menu bar is automatically attached underneath the form's title bar. When you first add a menu bar to the form, the menu bar is shown with a box with the text Item1. This is a first-level menu item (menu title). Replace this text with the title of your first menu.

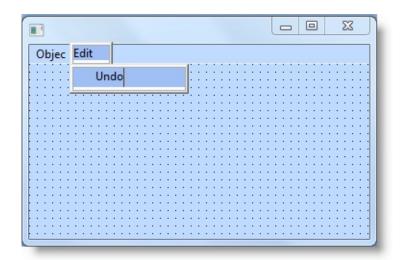


To add another menu title, use the Tab key:

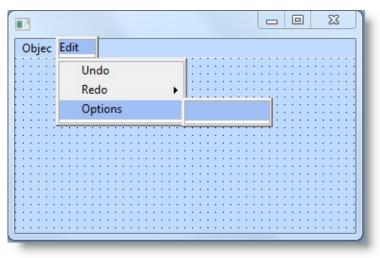
			23
Objec			

A box for the title of the second menu is displayed. Enter the caption for the second menu title.

To add menu items underneath the menu titles, press Enter and type the caption of the menu item in the edit box provided:



To add more items, use the Enter key. To add submenus, use the Tab key after you have typed in the caption:



To add more items in the submenu, press the Enter key.

Object	Edit	
	Undo	
	Redo 🕨	
	Options	User
		Network
		· · · · · · · · · · · · · · · · · · ·
:::::		
:::::		
		••••••

To move to the items and submenus to edit them, use the arrow keys.

To add a separator line, enter a dash (-).

If you look in the Source view, you will see that the editor has created the component definitions for the menu bar, the submenus and the menu items.

FUNCTION OPTIONS(\*DIRECT) BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) CLIENTHEIGHT(182) CLIENTWIDTH(354) HEIGHT(240) LEFT(353) MENUBAR(#MBAR\_1) TOP(235) WIDTH(370) DEFINE COM CLASS(#PRIM MBAR) NAME(#MBAR 1) PARENT(#COM OWNER) DEFINE COM CLASS(#PRIM MITM) NAME(#MITM 1) CAPTION('Object') DISPLAYPOSITION(1) PARENT(#MBAR 1) DEFINE\_COM CLASS(#PRIM\_SMNU) NAME(#SMNU\_1) PARENT(#MITM 1) DEFINE COM CLASS(#PRIM MITM) NAME(#MITM 2) DISPLAYPOSITION(1) PARENT(#SMNU\_1) DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_3) CAPTION('Edit') DISPLAYPOSITION(2) PARENT(#MBAR\_1) DEFINE\_COM CLASS(#PRIM\_SMNU) NAME(#SMNU\_2) PARENT(#MITM\_3) DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_4) CAPTION('Undo') DISPLAYPOSITION(1) PARENT(#SMNU 2) DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_5) CAPTION('Redo') DISPLAYPOSITION(2) PARENT(#SMNU\_2) DEFINE COM CLASS(#PRIM MITM) NAME(#MITM 6) CAPTION('Options') DISPLAYPOSITION(3) PARENT(#SMNU 2) DEFINE\_COM CLASS(#PRIM\_SMNU) NAME(#SMNU\_3) PARENT(#MITM\_6) DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_7) CAPTION('User') DISPLAYPOSITION(1) PARENT(#SMNU\_3) DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_8) CAPTION('Network') DISPLAYPOSITION(2) PARENT(#SMNU\_3) END\_COM

# 6.17.2 Accelerator Keys

An accelerator key for a menu title or option is the underlined letter of the caption. For example Object menu title can be accessed by pressing Alt + O.

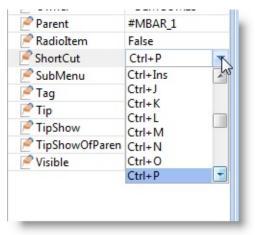
To indicate which letter is to be underlined, precede it with an ampersand (&). For example File is entered as &File.

Every menu item should have an accelerator key so that the menu can be used from the keyboard. Accelerator keys must be unique inside a submenu.

## 6.17.3 Shortcut Keys

A shortcut key is an additional keyboard shortcut. Typically it is a function key or a combination of Ctrl key and another key. For example a common shortcut key for save is Ctrl + S.

A shortcut is assigned using the ShortCut property of the menu item:



A shortcut key should be assigned only to the most commonly used menu options. A shortcut key is quicker to use than the accelerator key because it invokes the option directly, whereas when using accelerator keys the menu title has to be selected before selecting the option inside it.

Shortcut keys must be unique within the application.

## 6.17.4 A Few Things to Remember about Keyboard Access

Experienced users will very often use an application from the keyboard, not with the mouse. Therefore you need to make sure that every element of the interface can be easily and consistently accessed without the mouse. (Even though shortcut keys are used only in menu items, accelerator keys are also used, for example, in push-buttons, group boxes or checkboxes).

For menu titles and options, there are many standard Windows conventions for shortcut keys which should be followed. Below is a list of common shortcuts:

#### **Shortcut Function**

- Ctrl + N New menu option Ctrl + O Open
- Ctrl + P Print
- Ctrl + C Copy
- Ctrl + X Cut
- Ctrl + V Paste
- Ctrl + S Save
- Ctrl + Z Undo
- Ctrl + Y Redo

Note that F1 is commonly assigned for context sensitive help, Esc for Cancel and Enter for OK. Avoid assigning these keys to any other operations and be careful in setting the shortcut keys to avoid conflicts between different elements of the interface.

#### 6.17.5 Writing the Menu Events

You write the logic for the menu items in their Click event. For example, to make the Exit menu option close the application, you would write this code:

EVTROUTINE HANDLING(#MITM\_10.Click) exit ENDROUTINE

## 6.17.6 Creating a Standard Menu Automatically

To create a standard menu which you can customize, run the VL\_BBMNUBR template.

Choose Templates in the Design tab of the ribbon:

🗧 🛯 📂 🔚 迭 🧼 🕨 🔻 🛛 demcom01 - tes	st - LANSA Editor*						_
File Home Design Tools							
Design Theme Language	XDXStyles •	/ Style	Private	/ Mouseover	New Style	New Layout	ATLM_1
Design			Style			Layout Mar	nager - DEMCOM05

And then VL\_BBMNUBR:

Choose Temp	plate	X
VL_BASEMNT	Visual LANSA flat file maintenance	
VL_BASETAB	Visual LANSA tab folder with four sheets	
VL_BASETRE	Visual LANSA simple tree list	
VL_BASEWRK	Visual LANSA work with object	
VL_BBMNUBR	Visual LANSA: Menu bar building block	
VL_BBSTSBR	Visual LANSA: Status bar building block	
VL_BBWEBRT	Visual LANSA: Webroutine Building Block	
VL_FBASBOB	Visual LANSA Business Object Browser	
VL_FBASBOD	Visual LANSA Business Object Detailer	
VL_FBASBOF	Visual LANSA Business Object Filter	
VL_FBASBR1	Visual LANSA advanced parent browser	
VL_FBASBR2	Visual LANSA advanced tabbed detail	
VL_FBASMNT	Visual LANSA flat file maintenance	
VL_FBASTAB	Visual LANSA tab folder with four sheets	
VL_FBASTRE	Visual LANSA simple tree list	
VL_FBASWRK	Visual LANSA work with object	
VL_FBBMNUB	Visual LANSA: Menu bar building block	
VL_FBBSTSB	Visual LANSA: Status bar building block	*
Execute	Cancel Display All	

The template creates the menu without asking any questions.

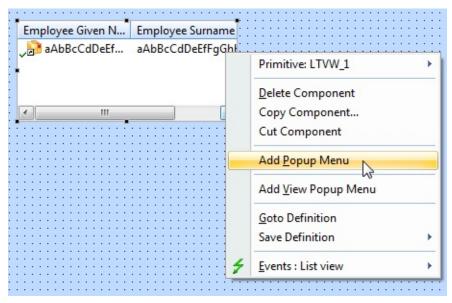
Help	
	· · · · · · · · · · · ·
· · · · · · · · · · · · · · · · · · ·	
· · · · · · · · · · · · · · · · · · ·	

When the menu has been created, you simply change the caption 'Object' to the name of the object your application deals with (invoice, employee, order etc.) and modify, delete and add items and submenus as appropriate.

# 6.17.7 Creating a Popup-Menu

Pop-up menus are displayed when the user selects a component and clicks the right mouse button. They are typically used to create shortcuts to options commonly associated with a component.

To create a pop-up menu open the Menus group in the Groups tab of the Repository tab. Drag a pop-up menu to the component you want to associate it with. Alternatively, right-click a component to bring up its pop-up menu and select the Pop-up Menu option. An edit box where you can enter the captions of the menu items is displayed on the top of the Design view.



Enter the captions of your menu items in the edit box. To add a item, press the Enter key.

Design S	ource	Reposito	ory Details   Repository Help   Cross References
Copy Delete			
			Employee Given N Employee Surname

At run time when the user right-clicks the field the pop-up menu is displayed:

Employee Given N	Employee Surname				
Veronica	Brown				
Fred	Bloggs				
Fred	Smithson	22	Сору		
Anne	Simpson	NS	Delete		
Shirley	Jones		Fritt		
John	Smvthe	*			

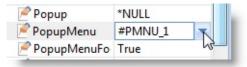
To edit the pop-up menu after you have created it, right-click the component it is associated with and choose the Edit Popup Menu option. Alternatively, select the pop-up menu in the Outline tab.

Note that when you are creating a pop-up menu for a field, you can add automatically enabled options for Cut, Copy and Paste. These options allow the end user to cut, copy and paste the value of the field.

If you want to use any of these automatic options, use the AutoActions property

of the pop-up menu. (In the current version only these three autoactions have been implemented and they only apply to fields).

The pop-up menu and the component are associated using the PopupMenu property of the component. You can use this property to assign a pop-up menu to another component:



Note that you can use the same pop-up menu with more than one component. For example in a explorer-type interface (a tree view on the left and a list view on the right), you would typically use just one pop-up menu:

#### DEFINE\_COM CLASS(#PRIM\_TRVW) NAME(#TREEVIEW) POPUPMEN DEFINE\_COM CLASS(#PRIM\_LTVW) NAME(#LISTVIEW) POPUPMEN

Then for every single menu item in the pop-up menu, you would write the code to perform an action depending on the type of item selected. In this example the tree view key columns are sourced from the #DEPTMENT and #SECTION fields, and the list view key column is sourced from the #EMPNO field:

```
EVTROUTINE HANDLING(#MiDetail.click)
if '#empno *ne *blanks'
 invoke #EmpForm.activateForm
 set #EmpForm emp_value(#empno) emp_dept(#deptment) emp_sect(#sectio)
else
if '#section *ne *blanks'
 invoke #SecForm.activateform
 set #SecForm dept_value(#deptment) sect_value(#section) top(#std_num)
else
if '#deptment *ne *blanks'
 invoke #DeptForm.activateform
 set #DeptForm dept_value(#deptment)
endif
endif
endif
ENDROUTINE
```

↑ 6.17 Menus

# 6.18 Displaying Messages and Errors

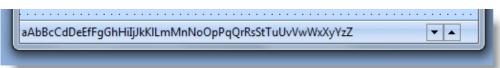
Messages can be displayed either in the status bar or in a message box. Display messages in the status bar when you want to show information about the status of an operation or information that does not require user response. Display a message box when you want to make sure the user sees the message or when you want input provided.

Fields, edit boxes, spin edit boxes and grid cells can change their background color to indicate an error. Fields and grid cells (because they are based on fields) automatically show error color when a field or file validation error is encountered. For edit boxes and spin edit boxes the error background color can be set programmatically using the ShowError property.

- 6.18.1 Status Bars
- 6.18.2 Message Boxes
- 6.18.3 Visually Showing Error States
- ↑ 6. Creating Applications Using Components

# 6.18.1 Status Bars

A status bar is attached to the bottom of a form.



A status bar can display:

- Information messages
- Status messages
- Validation messages

The easiest way to use the status bar for displaying messages is to run the template VL\_BBSTSBR.

For information about how to use the status bar refer to 6.14 Controls.

## **Clearing Errors**

You will notice that when you are executing a form which has a command button, the error messages are always cleared when the button is clicked. This is built-in behavior for a push-button.

Note that the I/O modules created before Visual LANSA Version 7.8 will have to be rebuilt for the forms to manifest this behavior.

Most other events have a value for the OPTIONS parameter set to \*NOCLEARMESSAGES as the default. This option stops errors being cleared by the event routine. A few events do not have this parameter, most notably the click event of a button.

To make an event to clear messages, set the OPTIONS parameter of the EVTROUTINE command of the event to \*CLEARMESSAGES.

↑ 6.18 Displaying Messages and Errors

### 6.18.2 Message Boxes

A message box is displayed on top of the current form. A message box is modal, in other words the user has to click one of the buttons in the message box to dismiss it before the underlying form is reactivated.

The message box consists of a title, an icon, message text and one or more command buttons as shown:



Message boxes are created using the MESSAGE\_BOX\_SHOW Built-In Function. Here is the source for the message box shown above:

use builtin(MESSAGE\_BOX\_SHOW) with\_args(OK OK Information 'Radio 1

The MESSAGE\_BOX\_SHOW function has the following arguments:

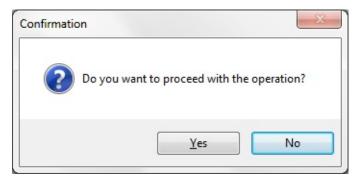
Argument	Description	Possible Values
Enable Buttons	The buttons shown in the message box.	YesNo YesNoCancel OK OKCancel AbortRetryIgnore RetryCancel
Default Button	The button that has the focus.	Any of the buttons in the Enable Buttons argument.
Icon to Show	The icon shown in the message box.	Question (question mark icon) Stop (stop icon) Exclamation (exclamation point

	icon) Information (information icon)
	Warning
	Asterisk
	Error
	Hand
	None
Title of Message Box	The text for the title enclosed in single quotes
Text to Show	The message text enclosed in single quotes

This code:

use builtin(MESSAGE\_BOX\_SHOW) with\_args(YesNo No Question 'Confirm

Creates and displays a message box with two buttons, Yes and No. No is the default button. The icon is a question mark as shown here:



### Getting the User's Response

When your message box has more than one button, you need to know which one the user has clicked. To get the response, use the TO\_GET parameter and specify a field where you want the response stored. The field must be an alpha field of 20 characters or less.

To store the response to the Confirmation dialog box in a field named #REPLY,

you would specify:

```
use builtin(MESSAGE_BOX_SHOW) with_args(YesNo No Question 'Confirm
```

You could then use the value of #REPLY to determine what action the program should take. For instance:

```
If cond'#reply *EQ Yes'
use builtin(MESSAGE_BOX_SHOW) with_args(OK OK Information " 'Yes
pressed')
else
use builtin(MESSAGE_BOX_SHOW) with_args(OK OK Information " 'No
pressed')
endif
```

Depending on what buttons the message box has, the return value can be:

- ABORT
- RETRY
- IGNORE
- OK
- CANCEL
- YES
- NO

### Make Sure You Use the Right Buttons

When deciding which buttons to put in the message box, think carefully about the text of the message to make sure that the text and the buttons match. For example if the message box asks a question, the correct buttons will be Yes, No and possibly Cancel, never OK and Cancel.

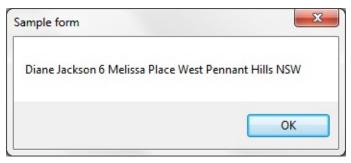
Also think about the context where the message is displayed. For instance, an information message should not have any other buttons than OK.

### Formatting the Message Text

The text of the message can be longer than in the previous examples and all or part of it can be retrieved from controls on the form (for instance fields or a grid) or from database. To create a message which consists of multiple text strings you can use either the MESSAGE\_BOX\_ADD or the MESSAGE\_BOX\_APPEND functions. The MESSAGE\_BOX\_APPEND function joins strings together whereas the MESSAGE\_BOX\_ADD adds the strings on separate lines.

# MESSAGE\_BOX\_APPEND

The MESSAGE\_BOX\_APPEND function joins strings together. The text in the dialog box below was created by appending the #GiveName #Surname #Address1 #Address2 and #Address3 fields of the PSLMST file together:



Here is the source:

change #empno 'A1234'

fetch fields(#givename #surname #address1 #address2 #address3) from\_file(ps

use builtin(MESSAGE\_BOX\_APPEND) with\_args(#givename) use builtin(MESSAGE\_BOX\_APPEND) with\_args(#surname) use builtin(MESSAGE\_BOX\_APPEND) with\_args(#address1) use builtin(MESSAGE\_BOX\_APPEND) with\_args(#address2) use builtin(MESSAGE\_BOX\_APPEND) with\_args(#address3) use builtin(MESSAGE\_BOX\_SHOW)

# MESSAGE\_BOX\_ADD

The MESSAGE\_BOX\_ADD function adds text strings on separate lines. This message box was created the same way as the example above but using MESSAGE\_BOX\_ADD statements.

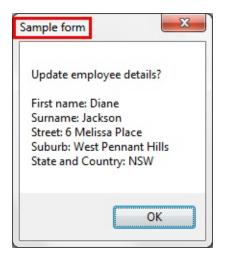


You can use up to ten items per line. For example you could add a question to the message box and static text before the fields to identify the rows:

change #empno 'A1234'

fetch fields(#givename #surname #address1 #address2 #address3) from\_file(ps

use builtin(MESSAGE\_BOX\_ADD) with\_args('Update employee details?') use builtin(MESSAGE\_BOX\_ADD) with\_args(') use builtin(MESSAGE\_BOX\_ADD) with\_args('First name: '#givename) use builtin(MESSAGE\_BOX\_ADD) with\_args('Surname: '#surname) use builtin(MESSAGE\_BOX\_ADD) with\_args('Street: ' #address1) use builtin(MESSAGE\_BOX\_ADD) with\_args('Suburb: '#address2) use builtin(MESSAGE\_BOX\_ADD) with\_args('State and Country: '#address3 use builtin(MESSAGE\_BOX\_SHOW)



If you do not specify any arguments in the MESSAGE\_BOX\_SHOW function, the form's description is used as the message box title, there is no icon and the

OK button is used.

With the formatting functions you can use all the arguments of the MESSAGE\_BOX\_SHOW function except the last one, Text to Show (if you specify any text, it will override the formatted text). The following statement adds Yes and No buttons, the question mark icon and the title Employee Details to the message box:

use builtin(MESSAGE\_BOX\_SHOW) with\_args(YesNo No Question 'Employ



1 6.18 Displaying Messages and Errors

# 6.18.3 Visually Showing Error States

A field and a grid cell will automatically change its background color when a field or file error is encountered. A validation error can be generated through an OAM or through the BEGINCHECK set of RDML commands.

mployee Details <u>Ski</u>	lls E <u>m</u> ail		
Employee Number	A0002		
Employee Surname	O'Toole		
Employee Given Name(s)	Neil		
Business Phone Number	9945667782		
Home Phone Number	958439044		
Start Date (DDMMYY)	000000		
Termination Date (DDMMYY)	000000		
Street No and Name			
Suburb or Town			
State and Country		Save	
Post / Zip Code	0	Cancel	
Employee Salary	.00	Delete	

## **Clearing Errors**

You will notice that when you are executing a form which has a command button, the error state is always cleared when the button is clicked in the same way as with error messages. This is built-in behavior for a push-button.

Most other events have a value for the OPTIONS parameter set to \*NOCLEARERRORS as the default. This option stops errors being cleared by the event routine. A few events do not have this parameter, most notably the click event of a button.

To make an event to clear errors, set the OPTIONS parameter of the EVTROUTINE command of the event to \*CLEARERRORS.

Note that when a field or a grid cell loses focus, the error background color is automatically cleared. This is built-in behavior that cannot be changed. This behavior is based on the assumption that when focus is moved away from the field or the cell, the error has been corrected. If it has not, the error state will be activated again the next time an event which contains commands which use validation rules is triggered.

# Showing Error States in Edit Boxes and Spin Edit Boxes

Edit boxes and spin edit boxes can be made to show error background color using the ShowError property. It should be remembered that it is always better practice to use fields and the repository-based validations than coding validation checking in the application.

# **Changing the Error Background Color**

The default error background color is Fuchsia. This color can be changed in two ways:

- By changing the ErrorBackgroundColor property in the VisualStyle of the component, or if the VisualStyleofParent property is set to True (as it normally should be) by changing the ErrorBackgroundColor of the parent component.
- By assigning a CLRREF value to the Registry setting "ERRORBACKGROUND" in the X\_LANSA key. This changes the error color for all components that either do not have a visual style or have specified the standard color 'Error' as the ErrorBackColor. This same color is also used by LANSA functions.

↑ 6.18 Displaying Messages and Errors

# **6.19 Multi-Form Applications**

This section describes how you create an application which consists of several forms. An application consisting of more than one form is called a multi-form application.

Important note: If you use the Visual LANSA Framework (Use *Framework* command on the *Tools* tab on the ribbon) to create applications, you do not need to know how to create multi-form applications. The Framework automatically handles all form interactions.

- 6.19.1 Basic Rules for Multi-Form Applications
- 6.19.2 A Two-Form Application
- 6.19.3 Create a Two Form Application
- 6.19.4 Show Multiple Instances of FRMDETAIL
- 6.19.5 Communication is always Between an Owner Form and a Member Form
- 6.19.6 Why Are the Contents of a Form Hidden from Other Forms?
- 6.19.7 Source Code for the Multi-Form Example Forms
- ↑ 6. Creating Applications Using Components

# 6.19.1 Basic Rules for Multi-Form Applications

When creating a multi-form application, there some basic rules and concepts you need to know:

- The forms of a multi-form application are always created as separate, independent forms
- The forms are joined together by choosing one of the forms as the main form of the application and then including the other forms in it. At run time the main form controls the other forms.
- A form cannot see what is inside another form (components or logic). All it knows about the other form are the properties, events and methods of the form itself.
- In most applications you use user-defined events, properties and methods for a form in addition to the standard ones. Typically the information that is custom-defined is application-specific, in other words it deals with the object the form works with, for example employees, invoices, departments.
- You create the user-defined events, methods and properties for the member forms, not the main form.

↑ 6.19 Multi-Form Applications

# 6.19.2 A Two-Form Application

The sample application used here has two windows (forms). The first window displays a list of employees, with their employee numbers and home phone numbers.

nployee				
New	ployee Giv	Employee Sur	Home Phone	
Details	ronica	Brown	(02) 9609 4627	
	v ed	Bloggs	344-2234454	
Exit	ed	Smithson	(02) 546-4657	
A0907	Anne	Simpson	090909	
A1001	Shirley	Jones	799 5268	
A1002	John	Smythe	047 629 0442	
A1003	Robert	Smithe	977 6268	
A1004	Ruth	Smithson	419 5656	
A1005	Peter	Smiths	674 4316	
A1006	Jack	Smithers	799 3638	
A1007	George	Snell	764 3562	
A1008	Alison	Sneddon	476 2198	
A1009	Damian	Snashall	605 8686	
A1010	Kate	Perry	684 1919	
A1011	Christopher	Perron	607 7587	
A1012	Patrick	Paul	687 1717	-

The application can show detailed information about the employee in a separate Employee Details window. This window is displayed when the user selects an employee and then the Details option from the Employee menu.

The Employee Details window shows the address details of the selected employee. It can also be used to change the details:

Employee Number	A1006	
Employee Given Name(s)	Jack	
Employee Surname	Smithers	
Street No and Name	8 Croydon Road,	
Suburb or Town	Croydon	
State and Country	NSW	
Post / Zip Code	2050	
Home Phone Number	799 3638	ОК
		Cancel

When the user changes the details of an employee on the Employee Details window and clicks the OK button, the details are updated in the database and the window is closed.

When this happens the list of employees in the Work with Employees window is updated to reflect the changes that were made in the Employee Details window.

↑ 6.19 Multi-Form Applications

# 6.19.3 Create a Two Form Application

This section describes how to create the application.

### **Step 1. Create Two Separate Forms First**

You started off by creating two separate forms. One was named WRKEMP and the other one FRMDETAIL. When the forms were created they were stored in the group Demonstration Material.

Name	WRKEMP
Description	Work with Employees
Group	Demonstration Material

It is easier to manage your application if you put all its forms in the same group or framework. Often you might want to create a group for the application.

You then design the user interfaces and the basic functionality for the forms in the editor:

- 1. Make the caption of WRKEMP 'Work with Employees' and create a list view with employee details. Add a menu with menu options.
- 2. Write the logic to fill the list in the Initialize event of the form.
- 3. Make the caption of FRMDETAIL 'Employee Details' and include several fields showing employee address information.
- 4. Add two command buttons. The Click event of the OK command button updates the employee details and closes the form; the Click event of the Cancel button just closes the form.

Now create the individual forms. This is always the first step of creating a multi-form application.

# Step 2. Include the Employee Details Form in the Work with Employee Form

Make WRKEMP the main form of the application. When you create a multiform application, you must have a main form. When the application is run, the main form can control the other forms in the application by setting their properties, invoking their methods and responding to their events.

Open WRKEMP in the editor. Then open the DEMONSTRATION group in the Groups tab of the Repository tab. This group shows all the forms that have been stored in this group.

Repository			
O Groups 🔹			
🔋 New 👻 📂 🗙	🕹   🔍 🕨 🗉		
ltem 🔺	Description	Modifi	
VI_BKOPN	Icon - Book open	*	
VI_CHECK	Icon - Check		
VI_CKIT	Icon - Click it		
VI_CROSS	Icon - Cross		
VI_DEPTCL	Icon - Department clo		
VI_DEPTOP	Icon - Department open		
VI_EMPLOY	Icon - Employee		
VI_EXEMPL	Icon - Ex Employee		
VI_FOLDCL	Icon - Folder closed		
VI_FOLDOP	Icon - Folder open		
TI_FOLDR1	Icon - Folder 1		
VI_FOLDR2	Icon - Folder 2		
VI_FOLDR3	Icon - Folder 3		
👕 VI_LANSA	Icon - LANSA		
TLOGO1	Icon - Logo 1		
TLOGO2L	Icon - Logo 2		
TI_SECTCL	Icon - Section closed		
TI_SECTOP	Icon - Section open		
VL_FLGAUS	Australian flag		
TL_FLGFRA	French flag		
VL_FLGGB	British flag		
T VL_FLGJPN	Japanese flag		
T VL_FLGNZ	New Zealand flag		
VL_FLGUSA	American flag	_	
WRKEMP	Work with Employee	17/	
> O DEVICES - Device	Device Components	*	
(		+	

Include the FRMDETAIL form in WRKEMP by dragging FRMDETAIL to the open form. A component definition statement is automatically added for it: DEFINE\_COM CLASS(#FRMDETAIL) NAME(#FRMDETAIL) HEIGHT(25)

You have now established the basic structure of the application: WRKEMP is the main form which owns FRMDETAIL.

WRKEMP is also called the owner form of FRMDETAIL. Similarly, FRMDETAIL can be referred to as a member form of WRKEMP.

FRMDETAIL is now shown as a member component of #WRKEMP in the Outline tab.

### **Owner Form and Member Forms**

When you include a form in an owner form, you are actually including an instance of the form. (For information about instances, refer to 6.8 Component Basics.). This means that:

- WRKEMP only contains a reference to FRMDETAIL. To edit FRMDETAIL you must open it in the editor.
- You can execute FRMDETAIL as a stand-alone form (although then it cannot communicate with WRKEMP). You can also include it in another form in another application.
- You can override the properties of FRMDETAIL when it is used inside WRKEMP and write code for its events. The changes you make to this instance do not change the FRMDETAIL component itself.
- You can display several instances of FRMDETAIL simultaneously. You have to do this if you want the user to be able to view the details of more than one employee at the same time. For more information, see 6.19.4 Show Multiple Instances of FRMDETAIL.

## Working with Multiple Forms

Now that you have created the forms, keep both of them open in the editor and moved between them using the Outline tab.

# **Establishing Communication between the Forms**

Establish how the two forms communicate with each other when the application is run. Owner forms can control the other forms using their properties, methods and events.

The owner form WRKEMP can:

- Set and retrieve the property values of FRMDETAIL. These properties can be the standard properties associated with forms or custom-defined properties.
- Invoke the methods of FRMDETAIL. The methods can be standard or userdefined.

• Respond to the events of FRMDETAIL. Again, the events can be standard events or user-defined events.

### WRKEMP cannot see what is inside FRMDETAIL

It is very important to note that WRKEMP can only use the properties, methods and events of the FRMDETAIL form itself, not of any of the fields or controls on it.

For example WRKEMP cannot set or retrieve the value of the #EMPNO field on FRMDETAIL, nor does it know about the OK or Cancel buttons.

### **Standard Form Properties, Events and Methods**

All forms have these standard properties, methods, and events:

Properties Height, Width, Top, Left, Caption etc.Methods ActivatForm, CloseForm, CloseFormQuery, HideForm,

- ShowForm, ShowModalForm MinimizeForm, MaximizeForm, RestoreForm
- Events Activate, Closing, CloseQuery, Deactivate, GotFocus, Initialize, LostFocus

For example the owner form WRKEMP form could use FRMDETAIL's standard properties, methods and events to:

• Move or resize FRMDETAIL or change its caption. This statement on WRKEMP will move FRMDETAIL:

SET com(#frmdetail) left(37) top(14)

- Show or hide FRMDETAIL. This statement will show it: invoke #frmdetail.ShowForm
- React somehow when FRMDETAIL is initialized or closed. This event routine monitors for its Closing event:

EVTROUTINE HANDLING(#frmdetail.Closing) << do something >> ENDROUTINE

However, of all of the above only the ShowForm and CloseForm methods were needed in our sample application.

Moreover, none of these standard properties, methods or events provide the means to establish the kind of communication between the forms that is required in our application. Therefore you define additional user-defined properties, methods and events to FRMDETAIL.

### **User-defined Properties, Events and Methods Were Required in FRMDETAIL**

User-defined properties, methods and events in FRMDETAIL were needed so that WRKEMP could:

• Pass the current employee number to FRMDETAIL

Data is typically exchanged between forms using properties, so add a property which communicates the employee number.

• Ask FRMDETAIL to fetch the details of the current employee

Actions are requested using methods, so add a method to FRMDETAIL which can be used to ask it to fetch the details.

• Know when the OK button on FRMDETAIL is clicked.

The fact that something has happened is signaled as an event, so add an event to FRMDETAIL.

Note that because the owner form WRKEMP can use the member form's properties, methods and events, all these changes are made to FRMDETAIL. No properties, methods, and events are added to WRKEMP because FRMDETAIL could not use them.

# **Step 3. Add User-defined Properties, Methods and Events to FRMDETAIL**

This section shows how the user-defined properties, methods, and events was added to FRMDETAIL.

## **3a. Adding Employee Property to FRMDETAIL**

To make it possible to pass the employee number from WRKEMP to FRMDETAIL, create a user-defined property called Employee in FRMDETAIL. Define it by adding the following statement:

Define\_pty name(Employee) set(\*auto #empno) get(\*auto #empno)

This statement specifies that FRMDETAIL has an Employee property which can

be set and retrieved by an owner form at run time. The SET parameter specifies that the value passed to this property is automatically assigned to the #EMPNO field on FRMDETAIL. The GET parameter specifies that when an owner form queries the value of the Employee property, the current value of field #EMPNO on FRMDETAIL is returned.

# **3b. Adding GetInfo Method to FRMDETAIL**

The user-defined method GetInfo was added to FRMDETAIL so that it could be asked to fetch the details of the current employee. This is the definition of the method in FRMDETAIL:

```
mthroutine name(GetInfo)
```

```
fetch fields(#detflds) from_file(pslmst) with_key(#empno)
endroutine
```

You may wonder why the FETCH statement wasn't put in one of the events in FRMDETAIL. The reason is that in our sample application none of the events of FRMDETAIL are suitable for executing the FETCH statement:

- The Initialize and Activate events are triggered before a value is passed to the Employee parameter
- The Changed event of the #Empno field is not triggered when the value is changed programmatically.

# **3c. Adding an OKPressed Event to FRMDETAIL**

To make it known that the OK button on FRMDETAIL has been clicked, a custom-defined event was specified on FRMDETAIL:

```
define_evt name(OKPressed)
```

It was specified that this event is signaled when the OK button is pressed on FRMDETAIL by adding a SIGNAL statement to the Click event of #OKBTN:

```
EVTROUTINE HANDLING(#OKBTN.Click)
signal event(OKPressed)
ENDROUTINE
```

So now every time the #OKBTN.Click event is triggered in FRMDETAIL it signals the OKPressed event to an owner form.

# The Modified FRMDETAIL

As a result of the modifications, FRMDETAIL now has a user-defined property, method and event which can be used by WRKEMP to communicate with it:

PropertyTo be used to set and retrieve the value of the #EMPNO field inEMPLOYEEFRMDETAIL.

Define\_pty name(employee) set(\*auto #empno) get(\*auto #ei

Method To be used to instruct FRMDETAIL to fetch the details of selec GETINFO employees. mthroutine name(GetInfo) fetch fields(#detflds) from\_file(pslmst) with\_key(#empno) endroutine

EventWhich tells other form that the OK button has been pressed onOKPRESSEDFRMDETAIL.

define\_evt name(OKPressed)
signal event(OKPressed)

You may want to have a look at the 6.19.7 Source Code for the Multi-Form Example Forms .

Even though these properties, methods and events were added on FRMDETAIL specifically to be used by the WRKEMP form in this sample application, they are, of course, available to any other form in which FRMDETAIL is included. In other words if you create another application in which FRMDETAIL is included, these properties, methods and events would be available to its owner form in that application.

### **Step 4. Modify WRKEMP to Communicate with FRMDETAIL**

Now WRKEMP is modified to communicate with FRMDETAIL.

## 4a. Showing FRMDETAIL

To display the FRMDETAIL form when the user chooses an employee from the list of employees on WRKEMP and then selects the Details option from the Employee menu, this statement is added to the Click event of the Details menu

```
item (MDETAILS) in WRKEMP:
  EVTROUTINE HANDLING(#MDetails.Click)
  SET COM(#frmdetail) formowner(#com_owner)
  invoke #frmdetail.ShowForm
  ENDROUTINE
```

The ShowForm method displays the nominated form and makes it the active window on the desktop.

Note that before showing the member form it is good practice to always expressly set its owner form using the run-time FORMOWNER property. This is because the first form that executes in an application becomes the owner of all member forms in the application by default, regardless of the basic structure of the application. In complex applications not setting the owner form may cause unpredictable results.

### 4b. Setting the Employee Property

```
We also wanted to set the value of the Employee property of FRMDETAIL as
soon as it is displayed. Therefore we added the SET statement in the Click
event of the Details menu item as well:
EVTROUTINE HANDLING(#MDetails.Click)
SET COM(#frmdetail) formowner(#com_owner)
invoke #frmdetail.ShowForm
set com(#frmdetail) employee(#empno)
ENDROUTINE
```

So now the current employee number is passed from WRKEMP to FRMDETAIL.

Even though we did not do this in the sample application, we could have set the value of the Employee property like this:

set com(#frmdetail) employee('A0070')

And we could also have retrieved the value of Employee like this: CHANGE FIELD(#EMPNO) TO(#FRMDETAIL.Employee)

## 4c. Getting the Details for the Current Employee

In addition to setting the value of the Employee property, we also wanted to instruct FRMDETAIL to fetch the details of the current employee. Again we

added the statement in the Click event of the Details menu item:

EVTROUTINE HANDLING(#MDetails.Click) SET COM(#frmdetail) formowner(#com\_owner) invoke #frmdetail.ShowForm set com(#frmdetail) employee(#empno) invoke #frmdetail.GetInfo ENDROUTINE

### 4d. Reacting to the OKPressed Event

To refresh the list of employees on WRKEMP when the OK button is pressed on FRMDETAIL, we created an event routine to monitor the OKPressed event:

EVTROUTINE HANDLING(#frmdetail.OKPressed) execute GetList ENDROUTINE

So now the GetList subroutine is executed on WRKEMP every time the OK button is pressed in FRMDETAIL.

You may want to have a look at the 6.19.7 Source Code for the Multi-Form Example Forms .

### A Thing to Note about Methods

Before we finish the example, it is worth noting that a form can invoke its own methods. Therefore we chose to put the CloseForm method of FRMDETAIL in the click event of its OK and Cancel buttons:

```
EVTROUTINE HANDLING(#CANCELBTN.Click)
invoke #frmdetail.Closeform
ENDROUTINE
```

```
EVTROUTINE HANDLING(#OKBTN.Click)
UPDATE FIELDS(#DETFLDS) IN_FILE(PSLMST) WITH_KEY(#EMPNC
invoke #frmdetail.Closeform
signal event(OKPressed)
ENDROUTINE
```

We could, of course, have invoked the method from WRKEMP, but in that case we would have had to create an event to signal that the Cancel button had been clicked.

# **Compiling the Forms**

The application was now complete. We saved both forms and then compiled them. We first saved and compiled FRMDETAIL so that its custom-defined properties became known. We then compiled WRKEMP.

# **Alternative Implementations**

We could have implemented the sample application in a slightly different way as well:

• The GetInfo method could have been modified to pass a value. In this case there would have been no need to define an Employee property for FRMDETAIL.

However, it is customary to define data as properties and actions as methods although in some cases the distinction is not clear-cut. Also, once a property has been defined, its value can also be queried by another form.

• It wasn't absolutely necessary to create the OKPressed event because we could have refreshed the list of employees in WRKEMP when FRMDETAIL was closed (Closing event). The drawback in this implementation is that the list would have been refreshed also when FRMDETAIL was closed using the Cancel button.

↑ 6.19 Multi-Form Applications

### **6.19.4 Show Multiple Instances of FRMDETAIL**

The application we created can only show the details of one employee at a time. Therefore we also created another version of the application which can show the details of more than one employee at a time:

nployee				
Employee N	lu Employee Giv Empl	oyee Sur Home Phone		
A0070	Veronica Brow	(02) 9609 4627		
A0090	Employee Details	and the constant		
A0193				
A0907	Employee Number	A0070		
A1001	Employee Given Nar	ne(s) Veronica		
A1002		Employee Details		- 0 <b>X</b>
A1003	Employee Surna	chipioyee betails		
A1004	Street No and Na	Employee Number	A0907	
A1005 A1006				
A1006 A1007	Suburb or Town	Employee Given Name(s)	Anne	
A1007 A1008	State and Count	Employee Surname	Simpson	
A1008	Post / Zip Code			
A1010	Post/ZipCode	Street No and Name	33 Anne street	
A1011	Home Phone Nu	Suburb or Town	Anneville	
A1012		State and Country	NSW	
		State and Country		
_		Post / Zip Code	2145	
		Home Phone Number	090909	OK
				Cancel

### Collections

To be able to display several instances of a form, we created a keyed collection of forms by adding this definition statement to WRKEMP:

DEFINE\_COM CLASS(#PRIM\_KCOL) NAME(#DFORMS) COLLECTS(#F

This statement defines a collection of instances of FRMDETAIL called #DFORMS. The form instances are identified by the employee number. We then changed the code to refer to an instance of the #DFORMS collection. The syntax for referencing a form instance is:

```
CollectionName<key>
```

This is what the code for the Details menu option now looks like:

EVTROUTINE HANDLING(#MDetails.Click) SET COM(#dforms<#empno>) formowner(#com\_owner) invoke #dforms<#empno>.showform set com(#dforms<#empno>) employee(#empno) invoke #dforms<#empno>.GetInfo ENDROUTINE

#### **Referring to All Form Instances**

The event routine for the OKPressed event in FRMDETAIL had to be changed to refer to any detail form in the collection:

EVTROUTINE HANDLING(#dforms<>.OKPressed) execute getlist ENDROUTINE

When the OKPressed event is signalled from any detail form, the GETLIST subroutine is executed. As you can see from this code, no key value is specified in the collection when all its instances are referred to: #dforms<>.

Lastly we added an event routine for the Closing event of WRKEMP so that all instances of FRMDETAIL forms are closed when the main form is closed:

```
EVTROUTINE HANDLING(#COM_OWNER.Closing)
invoke #dforms<>.CloseForm
ENDROUTINE
```

An alternative way of achieving the same result would have been to set FRMDETAIL's FormStyle to Owned. This setting specifies that if a form is a member form of another form, it will be closed when the owner form is closed.

You may want to have a look at the 6.19.7 Source Code for the Multi-Form Example Forms .

1 6.19 Multi-Form Applications

# 6.19.5 Communication is always Between an Owner Form and a Member Form

For two forms to exchange information, one form must be the owner form and the other one must be an instance of another form included in it, in other words a member form. Therefore:

- You could execute FRMDETAIL separately but in this case it could not communicate with WRKEMP. For the forms to be able to exchange information, you must execute the owner form and then invoke the member forms from it.
- If you included another form in WRKEMP, the FRMDETAILS form could not communicate with it directly. It could, of course, communicate with it through WRKEMP.
- A form you include in an owner form of the application can in turn be the owner of other forms. For example FRMDETAIL could be the owner of a Print Details form. FRMDETAILS could communicate with the Print Details form, but WRKEMP could not.
- Note that before showing the member form it is good practice to always expressly set its owner form using the run-time FORMOWNER property:

SET COM(#frmdetail) formowner(#com\_owner)

↑ 6.19 Multi-Form Applications

# **6.19.6** Why Are the Contents of a Form Hidden from Other Forms?

The contents of a form are hidden from other forms to reduce interdependencies between them. If an owner form could access the properties, events and methods of components in a member form, then changes in that component would require a change in the owner form as well.

Let's assume that a form has a field #Address1 which contains Street Name and Street Number. It also has a custom-defined property AddressInfo which other forms can use to set and retrieve the value of #Address1.

At some stage it becomes necessary to split the #Address1 field into two fields #StreetName and #StreetNumber. Consequently also the property routine for AddressInfo inside this form has to be changed. However, the AddressInfo property used by the other forms will not need to change. As a result, no change is required in the forms that set or get this property.

Similarly, you should consider creating user-defined methods to allow generalized processing for all owner forms. For example you can create a userdefined method for showing a form from an owner form. In its simplest, the method routine would be:

```
mthroutine uShowCustomerForm
invoke #com_owner.showform
endroutine
```

You could then add to this method routine code to set the size of the form, to activate it in case it is behind other windows, restore it if it is minimized. All this processing then becomes automatically available to any form that shows the Customer Form using the uShowCustomerForm method.

1 6.19 Multi-Form Applications

### 6.19.7 Source Code for the Multi-Form Example Forms

Work with Employees Form
Work with Employees Form with a Forms Collection
Employee Details Form
↑ 6.19 Multi-Form Applications

#### Work with Employees Form

FUNCTION OPTIONS(\*DIRECT); BEGIN\_COM CAPTION('Work with Employees'); ;

DEFINE\_COM CLASS(#PRIM\_LTVW) NAME(#LTVW\_1) DISPLAYPOSI DEFINE\_COM CLASS(#PRIM\_MBAR) NAME(#MBAR\_1) PARENT(#COI DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MEMPLOYEE) CAPTION DEFINE\_COM CLASS(#PRIM\_SMNU) NAME(#SMNU\_1) PARENT(#MEI DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MNEW) CAPTION(New) DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MDETAILS) CAPTION(D DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MSEPARATOR) CAPTIOI ) DISPLAYPOSITION(3) PARENT(#SMNU\_1);

DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MEXIT) CAPTION(Exit) I DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_1) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_2) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_3) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_4) DISPLAYPOSIT GROUP\_BY NAME(#ALLFLDS) FIELDS(#EMPNO #GIVENAME #SURN DEFINE\_COM CLASS(#FRMDETAIL) NAME(#FRMDETAIL) CAPTION(' ;

•

EVTROUTINE HANDLING(#COM\_OWNER.Initialize); execute getlist;

ENDROUTINE;

;

EVTROUTINE HANDLING(#MDetails.Click); Set com(#frmdetail) formowner(#com\_owner) invoke #frmdetail.ShowForm; set com(#frmdetail) employee(#empno); invoke #frmdetail.GetInfo; ENDROUTINE; ; EVTROUTINE HANDLING(#frmdetail.OKPressed);

execute getlist; ENDROUTINE;

;

```
subroutine name(getlist);
clr_list #ltvw_1;
select fields(#allflds) from_file(pslmst) ;
add_entry to_list(#ltvw_1);
endselect;
endroutine;
;
end_com;
```

↑ 6.19 Multi-Form Applications

### Work with Employees Form with a Forms Collection

FUNCTION OPTIONS(\*DIRECT);

BEGIN\_COM BORDERICONS(Maximize+Minimize+SystemMenu) CAPTI(;

DEFINE\_COM CLASS(#PRIM\_LTVW) NAME(#LTVW\_1) DISPLAYPOSI DEFINE\_COM CLASS(#PRIM\_MBAR) NAME(#MBAR\_1) PARENT(#COI DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MEMPLOYEE) CAPTION DEFINE\_COM CLASS(#PRIM\_SMNU) NAME(#SMNU\_1) PARENT(#MEI DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MNEW) CAPTION(New) DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MDETAILS) CAPTION(D DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MSEPARATOR) CAPTIOI ) DISPLAYPOSITION(3) PARENT(#SMNU\_1);

DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MEXIT) CAPTION(Exit) I DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_1) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_2) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_3) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_4) DISPLAYPOSIT GROUP\_BY NAME(#ALLFLDS) FIELDS(#EMPNO #GIVENAME #SURN DEFINE\_COM CLASS(#FRMDETAIL) NAME(#FRMDETAIL) BORDERIC DEFINE\_COM CLASS(#PRIM\_KCOL) NAME(#DFORMS) COLLECTS(#F

•

EVTROUTINE HANDLING(#COM\_OWNER.Initialize); execute getlist;

ENDROUTINE ;

;

EVTROUTINE HANDLING(#MDetails.Click);

Set com(#dforms<#empno>) formowner(#com\_owner)

invoke #dforms<#empno>.showform;

set com(#dforms<#empno>) employee(#empno);

invoke #dforms<#empno>.GetInfo;

ENDROUTINE ;

;

EVTROUTINE HANDLING(#dforms<>.OKPressed); execute getlist; ENDROUTINE ;

```
;
subroutine name(getlist);
clr_list #ltvw_1;
select fields(#allflds) from_file(pslmst);
add_entry to_list(#ltvw_1);
endselect ;
endroutine ;
;
EVTROUTINE HANDLING(#COM_OWNER.Closing);
invoke #dforms<>.CloseForm;
ENDROUTINE ;
end_com ;
;
```

↑ 6.19 Multi-Form Applications

### **Employee Details Form**

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Caption('Employee Details') C Define\_Com Class(#EMPNO.Visual) Name(#EMPNO) Displayposition(1) He Define\_Com Class(#GIVENAME.Visual) Name(#GIVENAME) Displaypositio Define\_Com Class(#ADDRESS1.Visual) Name(#ADDRESS1) Displaypositio Define\_Com Class(#ADDRESS2.Visual) Name(#ADDRESS2) Displaypositio Define\_Com Class(#ADDRESS3.Visual) Name(#ADDRESS3) Displaypositio Define\_Com Class(#ADDRESS3.Visual) Name(#ADDRESS3) Displaypositio Define\_Com Class(#POSTCODE.Visual) Name(#POSTCODE) Displaypositic Define\_Com Class(#PHONEHME.Visual) Name(#PHONEHME) Displaypositic Define\_Com Class(#PRIM\_PHBN) Name(#OKBTN) Caption('OK') Displaypositic Define\_Com Class(#PRIM\_PHBN) Name(#CANCELBTN) Caption('Cancel') Define\_Com Class(#PRIM\_STBR) Name(#STBR\_1) Displayposition(11) Hei Group\_By Name(#DETFLDS) Fields(#EMPNO #GIVENAME #SURNAME ;

\* define events and properties and methods

Define\_Evt Name(OKPressed)

Define\_Pty Name(employee) Get(\*auto #empno) Set(\*auto #empno) Mthroutine Name(GetInfo)

Fetch Fields(#DETFLDS) From\_File(PSLMST) With\_Key(#EMPNO) Endroutine

\* Event routines Evtroutine Handling(#CANCELBTN.Click) Invoke Method(#frmdetail.CloseForm) Endroutine Evtroutine Handling(#OKBTN.Click) Update Fields(#DETFLDS) In\_File(PSLMST) With\_Key(#EMPNO) Issue\_M Signal Event(OKPressed) Invoke Method(#frmdetail.CloseForm) Endroutine Endroutine End Com

1 6.19 Multi-Form Applications

### 6.20 Reusable Parts

Before attempting to understand reusable parts you should be familiar with multi-form applications. This is because you need to understand the concepts of user-defined properties, methods and events.

6.20.1 Why use Reusable Parts?

6.20.2 What Is Using a Reusable Part Like?

6.20.3 When Should Reusable Parts Be Used?

6.20.4 Create a Reusable Part

6.20.5 User-Designed Controls

↑ 6. Creating Applications Using Components

## 6.20.1 Why use Reusable Parts?

Reusable parts allow you to define components that can be reused in many different applications. The ability to quickly and easily reuse existing parts will yield significant benefits in the areas of:

- **Productivity** You don't have to reinvent the wheel. You can reuse logic that already exists and that has been tested.
- **Quality** Since reusable parts have been used before, they are much more likely to function first time than any new code that you have to write.
- **Consistency** Since many reusable parts have a visual aspect, reusing them will create a more consistent look and feel in your applications.
- **Maintainability** If you change (or correct) a reusable part it is instantly changed in all the applications that use it.

↑ 6.20 Reusable Parts

## 6.20.2 What Is Using a Reusable Part Like?

It's like using any other component such as a push button. A reusable part presents to the outside world:

- A set of properties which you can set and get just like the properties associated with a push button.
- A set of methods which you can invoke to request that the part performs some activity.
- A set of events which you can monitor for so that you are notified when something significant happens within the reusable part.

You are probably comfortable now with these concepts in the context of a simple component such as a push button or, for example, a list view.

When you create a list view, you must add the columns to it, load the details from a table, and then monitor for events indicating items in the list have been selected or deselected.

This is not really a complex task but there are two situations where you should create the list as a reusable part:

- If this department list is used on more than one form, you have to repeat the definition, loading and handling logic in all the different programs.
- If you want to change or enhance the list logic (or your logic has a defect in it) you have to change it in all the different programs.

Imagine you had a department list view defined in the repository as a reusable part:

Depa	Department Descr
ADM	Administration
AUD	Internal Auditing
FLT	Fleet Administrat
GAC	Group Accounts
INF	Information Servi
LEG	Legal
MIS	Mgmt. Informati
MKT	Marketing
SD	Sales & Distributi
TRVL	Travel

Imagine that it publishes a property named uCurrentDept (which tells you the department code currently selected) and it signals an event named

uDeptChanged (which tells you that a department selection has been made).

Imagine that in all other respects it looks after itself. It loads the list, handles the selection and deselection logic etc.

To you, a user of this reusable part, it is simply a black box (in other words an encapsulation) that creates a list of departments on your form, tells you when a department is selected (via the uDeptChanged event) and lets you query what the department code of the currently selected department is (via the uCurrentDep property).

Wouldn't this making designing and coding any application that needed to present a list of departments to the user much quicker, easier, more consistent and easier to maintain?

Finally imagine you have a large library of reusable parts and can assemble many of the things on your forms simply by drawing together pre-assembled and pre-tested components (reusable parts). The only code you need to write is to "glue" the reusable parts together in a logical manner.

↑ 6.20 Reusable Parts

## 6.20.3 When Should Reusable Parts Be Used?

Reusable parts enhance productivity in the following situations:

- **Simplification and Standardization** Whenever you suspect that the logic you are going to create can be simplified, standardized and reused in more than one form. The size and scale of the reusable part can vary from the stop/go lights on a task bar to complex prompting facilities.
- **Smart Visual Components** You can create "smart" visual components that you can reuse in hundreds of different forms.
- **Increased Modularity** If you have ever designed a form with a tabbed notebook containing 10 tabs you will realize that making the content of each tab into an individual component (even if it is not actually reused anywhere else) will make your life easier.
- **Compound Reusable Parts** As your skill in designing and reusing parts increases you will recognize that you can build compound reusable parts that are themselves constructed of other reusable parts

### Plan the Use of Reusable Parts

Reusable parts are a very useful and powerful productivity aid in almost all situations. However, you must think carefully whether a reusable part needs to be created because if you create too many reusable parts it becomes difficult to keep track of them.

Plan them carefully, document them properly and organize them consistently in appropriate groups and frameworks.

### **Use Reusable Parts in other Reusable Parts**

You should note that you can create reusable parts that contain other reusable parts.

↑ 6.20 Reusable Parts↑ 6.20 Reusable Parts

## 6.20.4 Create a Reusable Part

Note that you cannot execute a reusable part in isolation. It has to be imbedded in a form before you can observe and test its operation.

To be able to test your reusable part create a simple form like this:

Sample form	
Employee Surname	Brown
Employee Given Name(s)	Veronica
Department Code	INF
Section Code	DV
	Fetch
	Save

Follow these steps:

Step 1. Create your Test Form

Step 2. Create your Reusable Part

Step 3. Use your Reusable Part in your Test Form

Step 4. Add More Functionality to the Reusable Part

By the end of Step 4, you should realise the Benefits of Reusable Parts

↑ 6.20 Reusable Parts

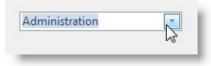
#### **Step 1. Create your Test Form**

- 1. Size the form and add the appropriate fields and buttons to it.
- 2. Add a status bar to your form (use the standard template to do this).
- 3. Add logic to your form so that the Fetch button fetches the required details from file PSLMST using the current value of field EMPNO (Employee Number) as the key.
- 4. Add logic to your form so that the Update button updates the employee details in file PSLMST for the selected employee.
- 5. Compile and execute your form. Verify that it functions as expected.

1 6.20.4 Create a Reusable Part

#### Step 2. Create your Reusable Part

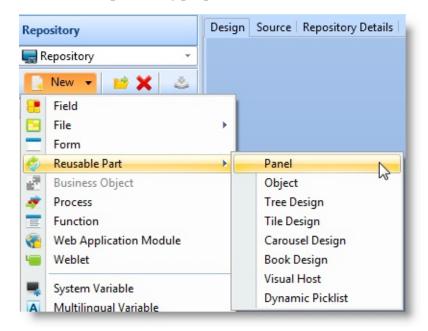
Next create a reusable part that looks like this:



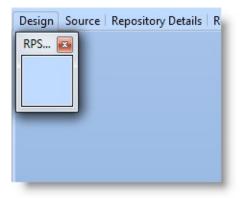
You will use it on your test form like this:

Sample form			×
Employee Surname Employee Given Name(s)	Brown Veronica		
Department Code Section Code	INF DV	Administration	
		Fetch	
		Save	

1. Create a new reusable part of type panel:

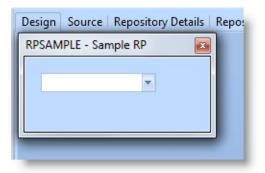


When the editor is displayed, the reusable part appears as a small panel:



This reusable part will eventually be embedded into other forms (or other reusable parts). The size that this panel has at the time that you save your reusable part determines the dimensions of the space that your reusable part will occupy on other forms (or in other reusable parts).

- 2. Size the panel and add a combo box to it.
- 3. Set the ComboBoxStyle property of the combo box to DropDownList.
- 4. Add columns to the combo box source from fields DEPTMENT (Department Code) and DEPTDESC (Department Description).
- 5. Make the DEPTMENT column in the combo box invisible. We only want the description details to be shown (field DEPTDESC). Your reusable part should look like this:



6. You need to fill the combo box with information. Add an Initialize event routine to your reusable part that fills the combo box with departmental details. For example:

```
EVTROUTINE HANDLING(#COM_OWNER.Initialize)
select (#deptment #deptdesc) from_file(deptab)
add_entry #CMBX_1
endselect
get_entry number(1) from_list(#CMBX_1)
```

#### ENDROUTINE

- This code fills the combo box from the DEPTAB file and then ensures that the focus is set to the first entry in the combo box.
- 7. Compile your reusable part. You cannot actually test your component yet. It first has to be embedded inside a form.

↑ 6.20 Reusable Parts

## Step 3. Use your Reusable Part in your Test Form

Edit the test form:

- 1. Using the Repository tab, locate your part and drag and drop it onto your form just as you would add a push button or a field.
- 2. Check the tabbing order of your form so that tabbing into your reusable part is correctly ordered. Recompile your form and execute it.
- 3. The reusable part should appear on your form and it should present a list of all department descriptions.

Sample form			
Employee Surname	Brown		
Employee Given Name(s)	Veronica		
Department Code	INF	Administration	-
Section Code	DV	Administration Internal Auditing Fleet Administration Group Accounts Information Services Legal Mgmt. Information Marketing	*

#### ↑ 6.20 Reusable Parts

## Step 4. Add More Functionality to the Reusable Part

At this stage, the departments reusable part is not that useful. It can display a list of departments and lets the user choose one, but nothing is done with the result. You actually need the reusable part to be able to:

- Change the department that is currently selected.
- Tell what department is currently selected.
- Tell when the selected department is changed.

Since all LANSA components have a standard property, event and method interface you will have to make your reusable part perform these required state changes and activities via these interfaces.

A single property named uCurrDepartment and a single event named uDepartmentChanged will do the job.

# Implement the uDepartmentChanged Event

1. Edit the reusable part so that whenever focus changes in the combo box it signals the uDepartmentChanged event. Define the event like this:

define\_evt uDepartmentChanged help('This event is signaled when the departu

2. Signal the event like this:

EVTROUTINE HANDLING(#COMBOBOX.ItemGotFocus) Signal uDepartmentChanged ENDROUTINE

## Implement the uCurrDepartment Property

Edit the reusable part to define property named uCurrDepartment which:

- Supports get operations by returning the value of field #DEPTMENT
- Supports set operations by setting the value of field #DEPTMENT and moving the focus in the combo box to the selected department.
- To do this you need to define the property like this: define\_pty name(uCurrDepartment) get(\*auto #deptment) set(SetDepartment)
- 2. And then add the property SetDepartment property routine like this: PTYROUTINE SetDepartment

```
define_map for(*input) class(#deptment) name(#sdeptment)
selectlist #combobox
continue if('#deptment *ne #sdeptment.Value')
set #ComboBox.CurrentItem Focus(True)
return
endselect
ENDROUTINE
```

3. Compile the new version of your reusable part.

## Use the New Properties and Events in your Form

1. Edit the test form so that when details of employee are fetched you need to tell the reusable part to show the details of the new department by adding a single line like this:

Set #II\_RVP29A uCurrDepartment(#deptment)

You also need to monitor for changes in the visual component. Whenever it signals uDepartmentSelected event you need to change the value of field #DEPTMENT using a routine like this:

EVTROUTINE HANDLING(#II\_RVP29A.uDepartmentChanged) Change #deptment #II\_RVP29A.uCurrDepartment ENDROUTINE

- 2. Finally, you should make field #DEPTMENT on your form ReadOnly(True) so that it cannot be changed via the user interface. In reality you would not normally even put field #DEPTMENT onto the form, but for purposes of this exercise it is useful because it allows you to see what is going on.
- 3. Recompile the test form.
- 4. Test your new version of the form. You should find that the reusable part switches to the correct department whenever you fetch details of an employee, and then whenever you select a department in the combo box that the value is instantly reflected in your form.

1 6.20 Reusable Parts

## **Benefits of Reusable Parts**

The power of Reusable Parts should now be quite evident. If the department reusable part had existed before your started to write your test form then you could have had:

- A better looking form (a combo box rather than an entry field).
- An easier to use form (choose from descriptions rather than key in a code).
- A standardized way of presenting and selecting departments.

#### ↑ 6.20 Reusable Parts

## 6.20.5 User-Designed Controls

A user-defined control is a special kind of reusable part for displaying list-type information. It offers more flexibility to the designer than predefined controls such as the tree view or list view controls.



User-designed controls can only be used with applications which are run in DirectX mode.

The Visual LANSA user-designed controls are:

- Carousel (prim\_caro)
- Tile (prim\_tile)
- Tree (prim\_tree)
- Book (prim\_book)

To see examples of the user-designed controls run the DirectX demonstration application (it is launched from the Partition Initialization dialog):



User-designed controls behave just like the other list controls, and obey the same list processing rules as tree view, list view and grid. For example they can be manipulated using the typical list commands such as Add\_Entry and Upd\_Entry, and respond to focus, selection, MouseOver and all the other typical events associated with visual list controls.

However, they have no defined appearance, in other words their design is userdesigned.

For a predefined list view (Prim\_ltvw) control, the definition in the source might appear as follows, with columns defined to indicate the data being used:

Define\_Com Class(#PRIM\_LTVW) Name(#LTVW\_1) Columnbuttonheight(19) Componentversion(2) Displayposition(1) Fullrowselect(True) Height(342) Keyboardpositioning(SortColumn) Left(9) Parent(#COM\_OWNER) Showsortarrow(True) Tabposition(1) Top(11) Width(680) Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Parent(#LTVW\_1) Source(#EMPNO) Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Displayposition(2) Parent(#LTVW\_1) Source(#SURNAME) Width(27)

Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Displayposition(3) Parent(#LTVW\_1) Source(#GIVENAME) Width(32) For a user-designed Tile (prim\_tile) the definition might be like this: Define\_Com Class(#prim\_Tile<#XDXTileDesign>) Name(#Tile) Displayposition(1) Height(359) Left(0) Parent(#COM\_OWNER) Tabposition(1) Top(30) Width(686)

Rather than columns, the tile simply has a design, in this case #XDXTileDesign. This defines the appearance of the item and any field values that will be made available to each item. See Designs.

## Designs

The visual design and the control need to interact. This means that such features as selection or focus appearance, which are built-in in predefined controls, need to be coded for user-designed controls.

To interact with the control the design needs to implement an interface:

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #XDXBasePanel *Implements
#Prim_Tile.iTileDesign *ListFields #ListFields) Height(60)
Layoutmanager(#Layout) Mouseoverstyle(#XDXStyles<MouseOver>)
Style(#XDXStyles<Item>) Width(248)
```

\* Fields received as on Add\_entry Group\_By Name(#ListFields) Fields(#Empno #Surname #Givename #Deptment #Section #Deptdesc #Secdesc)

The \*ListFields parameter, which can also be added to the Begin\_Com, defines the fields that will act as the underlying data for the list. The control passes the specified fields into the design instance at the moment an Add\_Entry command is executed.

For each of the events that can happen to a design, there is an equivalent OnXxxxx method on the interface that can be redefined within the design. This allows control to be passed to the design instance by the parent control. In this way, when the design is created by the use of Add\_Entry, the OnAdd method is executed:

```
Mthroutine Name(OnAdd) Options(*Redefine)
#lEmployee := ("&1 &2 (&3)").Substitute( #GiveName #Surname #Empno )
#lDepartment := ("&1 (&2)").Substitute( #Deptdesc #Deptment )
#lSection := ("&1 (&2)").Substitute( #Secdesc #Section )
Endroutine
```

Similarly, to manage focus being set, it is necessary to add a style to show focus, and when the item is no longer focus, the style needs to be removed:

```
Mthroutine Name(OnItemGotFocus) Options(*Redefine)
#Com_self.Styles.Add( #XDXStyles<Focus> )
Endroutine
Mthroutine Name(OnItemLostFocus) Options(*Redefine)
```

#Com\_self.Styles.Remove( #XDXStyles<Focus> ) Endroutine

Each individual user-designed control has its own interface. Most have similar methods for add, delete, focus and selection. Only tree is different because it also provides methods for expanding and collapsing items.

Refer to the Feature help (F1) for information about user-designed control properties, methods and events.

## **6.21 Ancestor Components**

Forms and reusable parts have a property called Ancestor. It is used to specify the component from which this component inherits from. "Inheriting" means the component has the same characteristics and contents as its ancestor, for example the same:

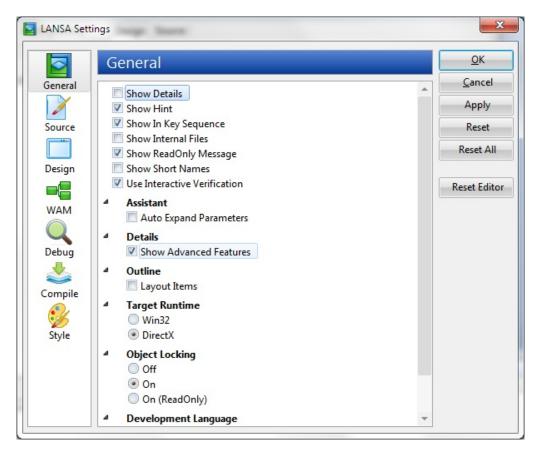
- Property settings such as Height, Width, Visible, Caption etc.
- Controls or other components on the ancestor form
- Event routines, subroutines, user-defined events, properties and methods as the ancestor.

By default forms inherit from #PRIM\_FORM which is the basic Visual LANSA form. It cannot be modified. However, you can specify any other form as the ancestor of a form.

By default reusable parts inherit from #PRIM\_PANEL. It is the basic Visual LANSA panel. You can make the ancestor of a reusable part to be any type of component (except a form), such as a button (#PRIM\_PHBN) or a LANSA field. If the reusable part contains only logic (and no visual interface), it is most memory efficient to make its ancestor #PRIM\_OBJ.

🔜 Reposi	tory		Ŧ
🖻 🗙	🕹 🔍 🕨 🗉	9	
Item	A	Description	
	📚 Meta pattern co	Meta pattern component	*
	<u>≷</u> Method compon	Method component reflects a Visua	
	🖄 Method paramet	Method parameter component refle	
	😨 Monitor Another	Informs a Target Field of the changi	
	🏉 Mouse compone	The mouse component indicates th	
	🔢 Multi-line Edit Box	Multi-line Edit Box has more than o.	
	🔢 Multi-line edit b	Multi-line edit box base class	
	┥ Number	Number Data Class	
	🍼 Number functio	Number function library	
	Numeric String	Numeric String Data Class	
_	💱 Object	Base object, use it as the ancestor of	
	Panel	Container for other components	
	📑 Pick List	Pick Lists are lists from which you c.	
1	= Pick List	Pick Lists are lists inside a property s	
	<ul> <li>Pick List Item</li> </ul>	Pick List Items are used in pick lists	
	🚱 Рорир	Popup	
1	📄 Pop-up Menu	Pop-up Menu	
	Primitive collecti	Primitive collection	
	💱 Primitive Compo	Primitive Component is not availabl	
	💱 Primitive Compo	Primitive Component is not availabl	
	💱 Primitive Compo	Primitive Component is not availabl	
	💱 Primitive Compo	Primitive Component is not availabl	
	Reimitive Compo	Drimitive Component is not available	-

The Ancestor and other Inheritance-related properties are only visible in the editor when the *Show Advanced Features* option is selected in the *LANSA Settings* dialog.



- 6.21.1 Form Ancestor
- 6.21.2 Button Ancestor
- 6.21.3 Date Display Ancestor
- 6.21.4 Advanced Inheritance Topics
- 1 6. Creating Applications Using Components

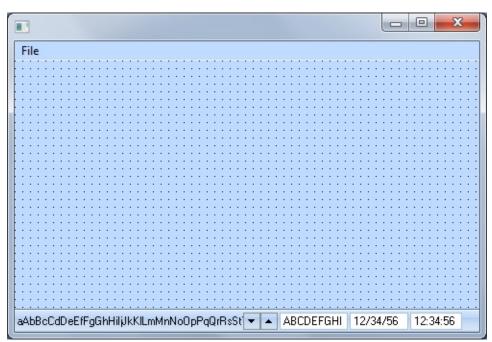
#### 6.21.1 Form Ancestor

When you create a new form and specify an ancestor for it, the inheriting form will look and behave exactly like its ancestor form.

Create an Ancestor Form Inherit from the Ancestor Change a Property in the Ancestor Override the Ancestor's Properties Redefine the Ancestor's Methods Customize the Inheriting Form Remove the Ancestor #STD\_FORM vs #PRIM\_FORM \$\overline{1} 6.21.1 Form Ancestor

### **Create an Ancestor Form**

To see how forms can inherit from each other, create a form to be used as the ancestor for other forms. The ancestor form will have a File menu and a status bar:



You can copy the Source for the Ancestor Form and paste it to a form and compile it. In this example we call the ancestor form EOEXAM01.

When EOEXAM01 is run, you can see it has a File menu with three options and a status bar showing the current user, date and time.

L		
File Open Close		
Exit		

The form contains a timer which updates the time every five seconds.

It also contains event routines for the File menu options which display message boxes. For example when the Exit option is selected the message box asks to confirm the action:

ANCESTOR		×
Are you sure	you want to clo	ose the form?
ſ	ОК	Cancel

Source for the Ancestor Form ↑ 6.21.1 Form Ancestor

#### Source for the Ancestor Form

\*\*\*\*\*\*\*

\*

\* COMPONENT: STD\_FORM

\*

\*\*\*\*\*\*

FUNCTION OPTIONS(\*DIRECT)

BEGIN\_COM LEFT(340) MENUBAR(#MBAR\_1) TOP(132) VISUALSTYI DEFINE\_COM CLASS(#PRIM\_STBR) NAME(#STBR\_1) DISPLAYPOSITI DEFINE\_COM CLASS(#USER.Visual) NAME(#USER) DISPLAYPOSITIO DEFINE\_COM CLASS(#DATE.Visual) NAME(#DATE) DISPLAYPOSITIO DEFINE\_COM CLASS(#TIME.Visual) NAME(#TIME) DISPLAYPOSITIO DEFINE\_COM CLASS(#PRIM\_TIMR) NAME(#TIMER) INTERVAL(5000) DEFINE\_COM CLASS(#PRIM\_MBAR) NAME(#MBAR\_1) PARENT(#COI DEFINE\_COM CLASS(#PRIM\_MBAR) NAME(#MITM\_1) CAPTION('File' DEFINE\_COM CLASS(#PRIM\_SMNU) NAME(#MITM\_1) PARENT(#MIT DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_FILE\_OPEN) CAP DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_FILE\_CLOSE) CA DEFINE\_COM CLASS(#PRIM\_MITM) NAME(#MITM\_FILE\_CLOSE) CA

\* Initialize the details on th status bar

EVTROUTINE HANDLING(#Com\_Owner.Initialize) OPTIONS(\*NOCLEAI Change #User \*User Change #Date \*Date Change #Time \*Time ENDROUTINE

\* Update the time on the status bar every 5 seconds (5000 milliseconds)

EVTROUTINE HANDLING(#TIMER.Tick) OPTIONS(\*NOCLEARMESSA Change #Time \*Time ENDROUTINE

\* Handle Exit request by attempting to close the form

EVTROUTINE HANDLING(#MITM\_EXIT.Click) OPTIONS(\*NOCLEARM

Invoke #Com\_Self.CloseForm ENDROUTINE

\* Handle file open menu item by invoking the "HandleFileOpen" method

EVTROUTINE HANDLING(#MITM\_File\_Open.Click) OPTIONS(\*NOCLE Invoke #Com\_Self.HandleFileOpen ENDROUTINE

\* Handle file close menu item by invoking the "HandleFileClose" method

EVTROUTINE HANDLING(#MITM\_File\_Close.Click) OPTIONS(\*NOCLE Invoke #Com\_Self.HandleFileClose ENDROUTINE

\* Handle a request to close the form by getting a confirmation

EVTROUTINE HANDLING(#Com\_Owner.Closing) OPTIONS(\*NOCLEAR Use Message\_Box\_Show (okCancel Ok Info #Com\_Self.Name 'Are you sure y If '#Std\_Obj = Cancel' Set #AfterCloseAction Value(None) Endif ENDROUTINE

\* Default method "HandleFileOpen". Typically this is redefined by a descenda

Mthroutine HandleFileOpen Use Message\_Box\_Show (ok Ok Info #Com\_Owner.Name 'Ancestor HandleF: Endroutine

\* Default method "HandleFileClose". Typically this is redefined by a descenda

Mthroutine HandleFileClose Use Message\_Box\_Show (ok Ok Info #Com\_Owner.Name 'Ancestor HandleF Endroutine

END\_COM

↑ Create an Ancestor Form

## Inherit from the Ancestor

Now create a new form (in this example we call it #EOEXAM02) which will inherit from the previously created ancestor #EOEXAM01:

- 1. Choose the Ancestor property of the new form and click on the button with three dots. A window showing all the forms in the repository is displayed.
- 2. Choose form #EOEXAM01. Click OK.

Now that you have specified the ancestor for the new form, it looks exactly like #EOEXAM01.

3. Compile and execute the new form. Notice that the status bar shows the same information as the ancestor form #EOEXAM01. The time is updated every 5 seconds. When you select any of the File menu options, the same message boxes are shown as when executing #EOEXAM01.

In other words the inheriting form looks and behaves exactly like its ancestor.

Sample form			x
File			
	PCXUSER	17/11/12	17:29:40

However, when you look at it source you will notice it does not contain any code. All its contents and its behavior are inherited from the ancestor form #EOEXAM01.

↑ 6.21.1 Form Ancestor

### **Change a Property in the Ancestor**

Any changes you make to the ancestor are immediately reflected in the form that inherits from it.

1. To see how this works, add accelerator keys (using an &) to the menu options in the ancestor form #EOEXAM01:



- 2. Compile the ancestor #EOEXAM01.
- 3. Next display the inheriting form #EOEXAM02. Notice that the menu accelerator keys are visible. Execute it. The accelerator keys are visible also at run time (press Alt to see them). Note that the changes in the inheriting form are visible immediately, there is no need to recompile it.

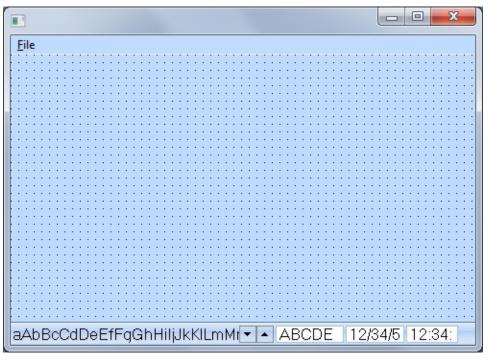
Sample form		
[ <u>F</u> ile]		
	PCXUSER	17/11/12 17:38:22

#### 1 6.21.1 Form Ancestor

## **Override the Ancestor's Properties**

If you change any properties in the inheriting form, these changes permanently override the settings it inherits from its ancestor.

For example the VisualStyle property of the status bar of the ancestor form #EOEXAM01 is \*NULL. If you change the visual style of the status bar in the inheriting form to #VS\_LARGE, this value overrides the VisualStyle setting of the status bar it had inherited.



A DEFINE\_COM statement is added to the source of the inheriting form which overrides the specification in the ancestor:

DEFINE\_COM CLASS(\*ANCESTOR) NAME(#STBR\_1) VISUALSTYLE(#

To revert back to the visual style of the ancestor #EOEXAM01, set the VisualStyle property of the status bar to \*NULL in the Details tab or delete the DEFINE\_COM statement.

It is worth remembering that any statements starting with DEFINE\_COM CLASS(\*ANCESTOR) indicate that this component is overriding properties it inherits from an ancestor.

↑ 6.21.1 Form Ancestor

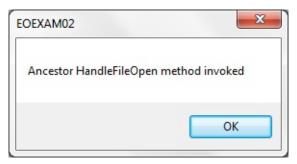
### **Redefine the Ancestor's Methods**

The ancestor form #EOEXAM01 contains a method routine which displays a message box when the Open option is selected in the File menu:

Mthroutine HandleFileOpen

Use Message\_Box\_Show (ok Ok Info #Com\_Owner.Name 'Ancestor HandleF: Endroutine

In the inheriting form exactly the same message is displayed when the Open option of the File menu is selected:

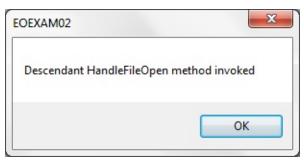


Sometimes you may want to modify the ancestor's methods in the inheriting form. You do this by using a method routine with \*REDEFINE option.

To see how this works add a method routine to the inheriting form #EOEXAM02 which overrides the HandleFileOpen method of the ancestor form by changing the message text to "Descendant HandleFileOpen method invoked":

MTHROUTINE NAME(HandleFileOpen) OPTIONS(\*REDEFINE) Use Message\_Box\_Show (ok Ok Info #Com\_Self.Name 'Descendant HandleF Endroutine

Compile the form and execute it. Select the Open option from the File menu. The redefined message text is displayed:



↑ 6.21.1 Form Ancestor

## **Customize the Inheriting Form**

The appearance and behavior set by the Ancestor form provide the basic look and behavior for the inheriting form.

You can then build on this basic structure by adding other components (in this example fields and a button) to the inheriting form. You can copy the Source for the Inheriting Form (below):

<u>F</u> ile		
Employee Number	ABCDE	Save
Employee Surname	aAbBcCdDeEfFgGhHiljJ	
Employee Given Name(s)	aAbBcCdDeEfFgGhHiljJ	
Street No and Name	aAbBcCdDeEfFgGhHiljJkKILm	
: ∶∶Suburb or Town	aAbBcCdDeEfFgGhHiljJkKlLm	
aAbBcCdDeEfFgGhHiljJkKlLmMnN	loOpPqQrRsSt - ABCDEFGHI 12/34.	/56 12:34:56

Compile and execute the form:

File Open Close Exit		Save
Employee Given Name(s) Street No and Name Suburb or Town		
	PCXUSER	17/11/12 17:46:08

The form now contains both inherited characteristics (the menu and status bar) and its own components and code.

When you look at the source of the inheriting form, you can see that it does not contain any DEFINE\_COM statements for the components defined in the ancestor (even though they are visible in the Outline tab) nor any of the routines specified in it. The inheriting form only contains component definitions and code specific to itself.

### Source for the Inheriting Form

\* and from which it inherits these behavioural characterstics: \*

- \* -> The user, date and time on the status bar are initialized
- \* -> The time on the status bar is updated every 5 seconds
- \* -> The menu bar "Exit" option is handled
- \* -> Closing of the form is handled
- \* -> Closing the form causes a confirmation to be displayed
- \* allowing the close to be cancelled.
- \* -> The "Open" option is handled (unless redefined, see following)
- \* -> The "Close" option is handled (unless redefined)

DEFINE\_COM CLASS(#EMPNO.Visual) NAME(#EMPNO) DISPLAYPOSI DEFINE\_COM CLASS(#SURNAME.Visual) NAME(#SURNAME) DISPLA DEFINE\_COM CLASS(#GIVENAME.Visual) NAME(#GIVENAME) DISPL DEFINE\_COM CLASS(#ADDRESS1.Visual) NAME(#ADDRESS1) DISPLA DEFINE\_COM CLASS(#ADDRESS2.Visual) NAME(#ADDRESS2) DISPLA DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#SAVE\_BUTTON) CAPTIC

\* Handle the save button ...

EVTROUTINE HANDLING(#Save\_Button.Click) Change (#Empno #Surname #GiveName #Address1 #Address2) \*Default Message 'Employee details have been saved' ENDROUTINE

\* Redefine the "HandleFileOpen" method so that this method is \* used instead of the ancestor's version of it ....

MTHROUTINE NAME(HandleFileOpen) OPTIONS(\*REDEFINE) Use Message\_Box\_Show (ok Ok Info #Com\_Self.Name 'Descendant HandleF: Endroutine

END\_COM

1 6.21.1 Form Ancestor

\*

#### **Remove the Ancestor**

If you want to clear the selected Ancestor for a form, set the Ancestor to the default ancestor #PRIM\_FORM.

↑ 6.21.1 Form Ancestor

# **#STD\_FORM vs #PRIM\_FORM**

When a form is created, it is always based on standard form #STD\_FORM (see 6.10.4 Modify Default Component Behavior). This standard form can be modified. For example the size or the visual style of the standard form can be changed, and this size and visual style become the default in all new forms.

The changes you make to #STD\_FORM only control the default values for new forms. There is no inheritance. Any changes you make to #STD\_FORM are not reflected in existing forms, they will only affect new forms.

Note that if you want your forms to inherit from a specific form by default, you can change the value of the Ancestor property of #STD\_FORM.

↑ 6.21.1 Form Ancestor

#### 6.21.2 Button Ancestor

When you create a new reusable part and specify an ancestor for it, the reusable part will look and behave exactly like its ancestor.

In this section we show you how to create a reusable part that has a button (#PRIM\_PHBN) as its ancestor.

Create a Button Reusable Part Use the Button Reusable Part Create another Button Reusable Part Use the Second Button Reusable Part

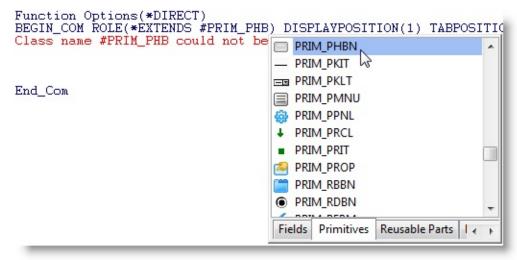
↑ 6.21 Ancestor Components

#### **Create a Button Reusable Part**

To see how reusable parts can inherit from a component such as a button, create a reusable part (in this example we call it EOEXAM20) of type panel, and make the ancestor of this part a button (#PRIM\_PHBN):

To see inheritance information, make sure that the *Show Advanced Features* option is selected in the *Editor Settings*.

Set PRIM\_PHBN as the Ancestor.



Next resize the component to make it look like a button:

EOEXAM20 - But.	. 💌

# **Set Button Properties**

Display the Details tab to see all the properties of the reusable part. Notice that it has now all the properties of a button (because its Ancestor is a button). Change the Caption to "Button A" and select Buttons as the Group property. The button reusable part now has a caption:

EOEXAM20 - But...

EOEXAM20 - But 🕰	
Button A	

#### **Create an Event Routine**

Lastly create an event routine for the Click event of the button to show a message box when it is clicked. The message box will show the name of the reusable part:

EVTROUTINE HANDLING(#COM\_OWNER.Click) Use Message\_Box\_Show (ok Ok Info 'The Button Is:' #Com\_Owner.Name) (# ENDROUTINE

Compile the reusable part.

#### Source for the Button Reusable Part

This is the source code for the button reusable part:

FUNCTION OPTIONS(\*DIRECT) BEGIN\_COM ROLE(\*EXTENDS #PRIM\_PHBN) CAPTION('Button A') DI:

EVTROUTINE HANDLING(#COM\_OWNER.Click) Use Message\_Box\_Show (ok Ok Info 'The Button Is:' #Com\_Owner.Name) (# ENDROUTINE

END\_COM

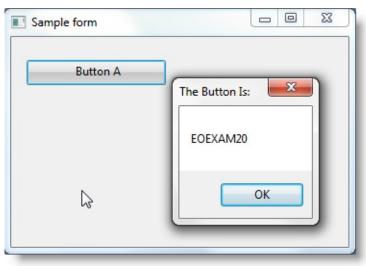
↑ 6.21.2 Button Ancestor

# **Use the Button Reusable Part**

Open a form. In the editor bring up the Repository tab, then the Groups tab and open the Buttons group.

Drag the button reusable part (EOEXAM20) to your form.

Compile and execute your form. The button with caption Button A is visible. When you click on it a message box showing the name of the button reusable part is displayed:



1 6.21.2 Button Ancestor

# **Create another Button Reusable Part**

The ancestor of a reusable part does not have to be a primitive component (#PRIM\_\*), it can also be another reusable part.

To see how this works, create a new reusable part (in this example we call it #EOEXAM21). Make its Ancestor the button reusable part (EOEXAM20) created in the previous section. Make its Caption "Button B".

Make the Group property of the part "Buttons".

Note that apart from the BEGIN\_COM statement your reusable part does not contain any code:

FUNCTION OPTIONS(\*DIRECT) BEGIN\_COM ROLE(\*EXTENDS #EOEXAM20) CAPTION('Button B') END\_COM

However, even though not visible in the source code, it inherits the Click event behavior from its ancestor EOEXAM20.

Compile the reusable part.

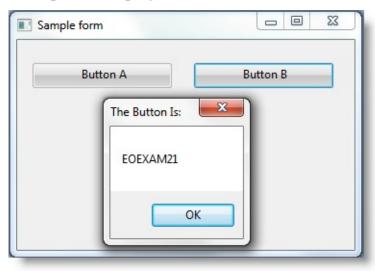
↑ 6.21.2 Button Ancestor

#### **Use the Second Button Reusable Part**

Open a form. In the editor bring up the Repository tab, then Groups tab and open the Buttons group.

Drag the second button reusable part (EOEXAM21) to your form.

Compile and execute your form. The EOEXAM21 button with its caption "Button B" is visible. When you click on Button B a message box showing the name of the reusable part is displayed:



Note that the routine for the Click event which displays the message box is inherited from the ancestor EOEXAM20.

```
↑ 6.21.2 Button Ancestor
```

#### 6.21.3 Date Display Ancestor

In this section we first show you how to create a reusable part which validates and formats dates, and then how to use this reusable part as the ancestor of another reusable part which extends the date validation functionality to indicate whether the date is a weekday.

Create a Date Display Reusable Part Use the Date Display Reusable Part Create another Date Display Reusable Part Redefine the Ancestor's Methods Use the Inheriting Date Display Reusable Part  $\widehat{}$  6. Creating Applications Using Components

# **Create a Date Display Reusable Part**

Create a reusable part (in this example we call it EOEXAM04). Copy and paste the code from the source code supplied below.

Set its Group property to Dates & Times then compile it.

The reusable part contains an input field in which a date can be entered and a label that will display the formatted date:

EOEXAN	104 - Sample RP	
Date	ABCDEF	<text></text>

When the part is executed (in a form) it will look like this:

From Date (MMDDYY) 111712 17th NOVEMBER 2012

The part has three user-defined properties:

uCaption Is used to the set the caption for the date field.

uDate Is used to set and validate the entered and to retrieve it.

UInError Is set to True if the date cannot be validated.

# Source for Date Display Reusable Part

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #PRIM\_PANL) DISPLAYPOSITION(1) HEI DEFINE\_COM class(#DATEC.Visual) name(#SHOWDATE) CAPTION('Date DEFINE\_COM class(#PRIM\_LABL) name(#SHOWTEXT) CAPTION('<text

DEFINE field(#RETCODE) type(\*CHAR) length(1) desc('Return Code') DEF\_COND name(\*OKAY) cond('#RetCode = Y') DEFINE field(#CVTDATE) type(\*CHAR) length(20)

\* Published Properties

DEFINE\_PTY name(uCaption) set(Set\_uCaption) DEFINE\_PTY name(uDate) get(Get\_uDate) set(Set\_uDate) DEFINE\_PTY name(uInError) get(Get\_uInError) set(Set\_uInError)

\* Property Handling routine

PTYROUTINE name(Set\_uCaption) DEFINE\_MAP for(\*input) class(#Std\_TextL) name(#Property\_001) SET com(#ShowDate) CAPTION(#Property\_001.Value) ENDROUTINE

PTYROUTINE name(Set\_uInError) DEFINE\_MAP for(\*input) class(#Std\_Bool) name(#Property\_002) SET com(#ShowDate) SHOWERROR(#Property\_002.Value) ENDROUTINE

PTYROUTINE name(Get\_uInError) DEFINE\_MAP for(\*Output) class(#Std\_Bool) name(#Property\_002) SET com(#Property\_002) VALUE(#ShowDate.ShowError) ENDROUTINE

PTYROUTINE name(Set\_uDate) DEFINE\_MAP for(\*input) class(#DateC) name(#Property\_003) SET com(#ShowDate) VALUE(#Property\_003.Value) INVOKE method(#Com\_Self.ValidateDate) ENDROUTINE

PTYROUTINE name(Get\_uDate) DEFINE\_MAP for(\*Output) class(#DateC) name(#Property\_003) SET com(#Property\_003) VALUE(#ShowDate.Value) ENDROUTINE

\* Handle change of date by validating its value

EVTROUTINE handling(#SHOWDATE.Changed) options(\*NOCLEARMES: INVOKE method(#Com\_Self.ValidateDate) ENDROUTINE

\* Method to validate the date

MTHROUTINE name(ValidateDate)

SET com(#Com\_Owner) UINERROR(FALSE)

CHANGE field(#DATEC) to('#SHOWDATE.VALUE')

IF\_NULL field(#DATEC)

SET com(#ShowText) CAPTION('No date specified')

ELSE

USE builtin(CONVERTDATE) with\_args(#DATEC F Q) to\_get(#CVTDATE #

```
IF cond(*OKAY)
SET com(#ShowText) CAPTION(#CvtDate)
ELSE
SET com(#Com_Owner) UINERROR(TRUE)
SET com(#ShowText) CAPTION('Invalid Date')
ENDIF
```

ENDIF

ENDROUTINE

END\_COM

1 6.21.3 Date Display Ancestor

#### **Use the Date Display Reusable Part**

Open a form. In the editor bring up the Repository tab, then the Groups tab and open the Dates and Times group. Drag the date display reusable part (EOEXAM04) to your form.

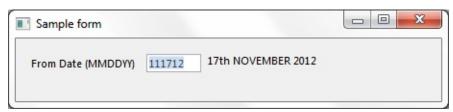
Compile and execute your form and enter a date in the input field. Note that the dates must be entered in the format mmddyy.

Sample form		
Date	111112 11th NOVEMBER 2012	

The reusable part has a uDate property which you can use to set the date and a uCaption property which you can use to set its caption. Add this code to the form:

```
EVTROUTINE handling(#COM_OWNER.Initialize)
SET com(#EOEXAM04) UDATE(*mmddYYc) UCAPTION('From Date (MM
ENDROUTINE
```

Compile and execute the form. The caption of the reusable part will now be From Date (MMDDYY) and the date will default to the current date:



1 6.21.3 Date Display Ancestor

# **Create another Date Display Reusable Part**

You can create another reusable part which extends the functionality of the date display reusable part #EOEXAM04.

To see how this works, create a reusable part (in this example we call it #EOEXAM05). Make its Ancestor the date display reusable part (EOEXAM04) created in the previous section.

Make the Group property of the part "Dates & Times".

Note that apart from the BEGIN\_COM statement your reusable part does not contain any code:

FUNCTION options(\*DIRECT)

```
BEGIN_COM role(*EXTENDS #EOEXAM04) HEIGHT(29) WIDTH(492)
END_COM
```

However, even though not visible in the source code, it inherits all the functionality of its ancestor. (If you want to test this, compile the reusable part, put it in a form and compile and execute the form.)

↑ 6.21.3 Date Display Ancestor

#### **Redefine the Ancestor's Methods**

To extend the functionality of the date display reusable part #EOEXAM04, modify the ancestor's methods in the inheriting reusable part. You do this by using a method routine with \*REDEFINE option.

The ancestor date display reusable part #EOEXAM04 contains a method routine ValidateDate which validates the dates. We want to extend the date validation to ensure that the day is a week day.

To see how this works add a method routine to the inheriting reusable part #EOEXAM05 which overrides ValidateDate method in the ancestor:

MTHROUTINE name(ValidateDate) options(\*REDEFINE) ENDROUTINE

Make the redefined validation routine check whether the date is a weekday and show the day of the week. You can copy this code from the Source for Inheriting Date Display Reusable Part below.

Compile and execute your reusable part.

#### Source for Inheriting Date Display Reusable Part

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #EOEXAM04) HEIGHT(29) WIDTH(492)

\* Override ancestor date validation add additional rules

\* to ensure that the date is not just a valid date but

\* also a valid day of the week .....

MTHROUTINE name(ValidateDate) options(\*REDEFINE) DEFINE field(#RETCODE) type(\*CHAR) length(1) desc('Return Code') DEF\_COND name(\*NOTOKAY) cond('#RetCode \*ne Y') DEFINE field(#CVTDATE) type(\*CHAR) length(20)

\* Do standard ancestor validaton INVOKE method(#Com\_Ancestor.ValidateDate)

\* If a date exists and it is valid so far, check that it is a week day IF cond('(#Showdate.Value \*ne \*Blanks) and (#Com\_Owner.uInError = False)

CHANGE field(#CVTDATE) to(\*NULL) CHANGE field(#DATEC) to('#SHOWDATE.VALUE') USE builtin(CONVERTDATE) with\_args(#DATEC F R) to\_get(#CVTDATE #

CASE of\_field(#CVTDATE) WHEN value\_is('= Mon' '= Tue' '= Wed' '= Thu' '= Fri') USE builtin(CONVERTDATE) with\_args(#DATEC F S) to\_get(#CVTDATE # USE builtin(TCONCAT) with\_args(#SHOWTEXT.CAPTION ' (' #CVTDATE OTHERWISE SET com(#Com\_Owner) UINERROR(TRUE) USE builtin(TCONCAT) with\_args(#SHOWTEXT.CAPTION ' (Not a week date) ENDCASE

SET com(#ShowText) CAPTION(#Std\_Textl) ENDIF ENDROUTINE

END\_COM

1 6.21.3 Date Display Ancestor

#### **Use the Inheriting Date Display Reusable Part**

Open a form. In the editor bring up the Repository tab, then the Groups tab and open the Dates and Times group.

Drag the inheriting date display reusable part (EOEXAM05) to your form. Compile and execute your form and enter a date in the input field. The dates must be entered in the format mmddyy.

The formatted date shows the day of the week:

Sample form		
Date	010101 1st JANUARY 2001 (MONDAY)	

If the date is not a a weekday, an error is displayed:

Sample form		
Date	010111 1st JANUARY 2011 (Not a week day!)	

As you can see, by redefining the date validation method the inherting reusable part extends the functionality of its ancestor.

1 6.21.3 Date Display Ancestor

#### **6.21.4 Advanced Inheritance Topics**

Inherit User-defined Events, Methods and Properties
Hide User-defined Events, Methods and Properties
#Com\_Owner, #Com\_Ancestor and #Com\_Self
↑ 6. Creating Applications Using Components

# Inherit User-defined Events, Methods and Properties

In this section three sample forms are used:

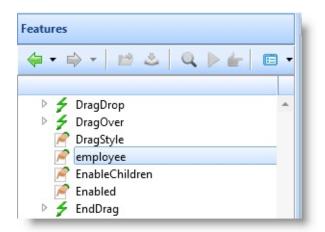
Form A -FormA defines a protected event OKPressed and a private (hidden) Ancestor Component Begin Com Protect(OKPressed) Private(GetInfo) Role(\*EXTE) Form B -FormB inherits from FormA and makes the OKPressed event publ Inheriting Begin Com Public(OKPressed) Role(\*EXTENDS #FORMA) Component Form C -FormC owns FormB and contains an event routine to handle the C **Owner** Define\_Com Class(#FORMB) Name(#FORMB) Component evtroutine handling(#FORMB.OKPressed) endroutine

When you create user-defined events, properties and methods for a form, by default they will be visible to other forms that will use this form as an ancestor.For example a form (#FRMDETAIL) has a user-defined property Employee:Define\_pty name(Employee) set(\*auto #empno) get(\*auto #empno) help('Use

```
a custom-defined event OKPressed:
define_evt name(OKPressed)
```

```
and a custom-defined method GetInfo:
    mthroutine name(GetInfo)
    fetch fields(#detflds) from_file(pslmst) with_key(#empno)
    endroutine
```

The OKPressed event is visible in the Events tab of the Details tab. The Employee property is visible in the Features tab:



And the GetInfo method is visible in the Methods tab of the Details tab.

If you now specify this form (FRMDETAIL) as the ancestor of another form, the code used to define the user-defined event, method and property is not visible, but the event, property and method are visible through the Details tab or the help and you can use them in the code of the inheriting form just as you would in the ancestor form.

↑ 6.21.4 Advanced Inheritance Topics

# Hide User-defined Events, Methods and Properties

If an event, method or property is only meant to be used inside an ancestor form or reusable part, it can be hidden from all inheriting components using the Private property of the form or the reusable part.

Events, methods and properties defined as private are hidden from both inheriting and owner forms and reusable parts.

For example, to hide the GetInfo method, select the Private property of the ancestor form and select GetInfo from the list of possible values:

🖻 PopupMenu	*NULL		
🖻 Private	GetInfo		
PrivateStyle	employee		
Protect	GetInfo		
Public 🖉	OKPressed 13		
RenderStyle	Application		

Save the form. The GetInfo method is now hidden from all other components.

Display FormB and bring up the Methods tab. GetInfo is no longer visible which indicates it is no longer available in FormB.

Note however that hidden properties, events and methods are still visible in the Feature help.

# Partly Hide User-defined Events, Methods and Properties

If you want to hide an event, method or property from owner forms and reusable parts but still make them available to inheriting forms and reusable parts, define the event, property or method as Protected.

For example, to protect the OKPressed event, select the Protect property of the ancestor form and select OKPressed from the list of possible values:

PrivateStyle	*NULL	
Protect	OKPressed	-
Public	employee GetInfo	
RenderStyle	OKPressed	
ScalingFactorHor	Jooo	1
ScalingFactorVer	9600	Ŭ

Save the ancestor form. The OKPressed event is now hidden from owner components, but it is available to inheriting forms and reusable parts. Display the inheriting form and bring up the Events tab. OKPressed is still

visible which indicates it is available.

Then include the ancestor form (FRMDETAIL) in another form. Display the Events Tab for FRMDETAIL as a member form. The OKPressed event is not visible.

Note however that hidden properties, events and methods are still visible in the Feature help.

# Make Partly Hidden User-defined Events, Methods and Properties Visible

In some circumstances it is necessary to make public user-defined events, methods and properties which have been hidden in the ancestor from owner components using the Protect property. This can be done using the Public property of the inheriting form or reusable part.



Note that the system-internal CreateInstance and DestroyInstance values are always displayed for Public. Do not use them.

Now include FormB in another form. Display the Events tab. The OKPressed event is visible and available to the owner form.

↑ 6.21.4 Advanced Inheritance Topics

# #Com\_Owner, #Com\_Ancestor and #Com\_Self

The generic name #Com\_Owner can be used to refer to the current component. In the context of inheritance there are two other generic names you can use for components: #Com\_Ancestor and #Com\_Self.

#Com\_Ancestor is a generic reference to the ancestor of the current component and #Com\_Self is a generic reference to the currently executing component.

# #Com\_Owner and #Com\_Self

To see in what way #Com\_Owner and #Com\_Self differ create two reusable parts with this code:

**Button** This button reusable part contains a method MethodOne which display:

3 message is displayed when this button (Button3, in other words #Com\_ FUNCTION options(\*DIRECT)

BEGIN\_COM role(\*EXTENDS #PRIM\_PHBN) CAPTION('Button

MTHROUTINE name(MethodOne) USE builtin(MESSAGE\_BOX\_SHOW) with\_args(OK OK INFO " 'I ENDROUTINE

EVTROUTINE handling(#COM\_OWNER.Click) INVOKE method(#com\_owner.methodone) ENDROUTINE

END\_COM

Button4 This button reusable part inherits from Button3 and redefines MethodO4 Bye.

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #BUTTON3) CAPTION('Button 4')

MTHROUTINE name(MethodOne) options(\*redefine) USE builtin(MESSAGE\_BOX\_SHOW) with\_args(OK OK INFO " 'I ENDROUTINE

#### END\_COM

Add these buttons to a form and compile and execute the form. Click on both buttons. They both display a message saying 'Hello':

				<u>_                                    </u>
Button 3			Button 4	
		×		
	Hello			
	OK			

In other words the redefined method which would display the message 'Bye' is not used for Button 4 because the Click event defined in Button 3 invokes the method in the component where it is defined (#Com\_Owner):

INVOKE method(#com\_owner.methodone)

Next change the invoke command in the Click event of Button 3 to use #Com\_Self instead of #Com\_Owner so that the Click event invokes the method in the component where the event is executed:

INVOKE method(#com\_self.methodone)

Compile Button 3 and execute your form again. This time when you click on Button 4, the redefined method is used and the message 'Bye' is displayed:

	×
Вуе	
(OK	

#### #Com\_Ancestor

If you want to invoke a method in an ancestor component, you can refer to it by qualifying the name of the method using the generic name #com\_ancestor.

For example, to continue with the above example, you can invoke the MethodOne method in Button 3 from the redefined MethodOne in Button 4 like

this:

MTHROUTINE name(MethodOne) options(\*redefine) INVOKE method(#com\_ancestor.methodone) USE builtin(MESSAGE\_BOX\_SHOW) with\_args(OK OK INFO " 'Bye') to\_g ENDROUTINE

Compile Button 4 and run your test form. When you click on Button 4 first the MethodOne from the ancestor Button 3 is invoked:

<

And then when you click OK, the Message\_Box\_Show Built-In Function showing the text 'Bye' defined in Button 4 is displayed:



↑ 6.21.4 Advanced Inheritance Topics

## 6.22 Collections

Collections are just what their name implies: a grouping of components of the same type.

In its simplest form a collection may be thought of as an array or a list. For example, you can create a collection of employee names (field #FULLNAME) indexed by a number like this:

Define\_Com Class(#PRIM\_KCOL<#FullName #Std\_Num>) Name(#Employe

Collections are much smarter than arrays because rather than being indexed by a number they can be keyed by anything you choose. For example you can create a collection of employee salaries that is keyed by their names like this:

Define\_Com Class(#PRIM\_KCOL<#Salary #FullName>) Name(#Employees)

The power of collections start to become obvious when you use them to collect more complex objects than just simple names and salaries. For example, imagine you had a form named FORMA. In another form named FORMB you can create a collection of FORMAs like this:

Define\_Com Class(#PRIM\_KCOL<#FORMA #Std\_Num>) Name(#ChildForr

By keeping a collection of child forms like this you can easily manage them singly or as a whole group.

Repository			
🍰 Primitives			
🖻 🗙 🕹 🔍 I			
Item 🔺	Description	Modified	Build :
🗾 Image	Shows a picture file		-
Interface base class	Interface base class		
🧿 Interface proxy base	Interface proxy base cl		
<ul> <li>Intrinsic function lib</li> </ul>	Intrinsic function library		
🐚 Keyboard component	The keyboard compo		
Keyed Collection	A set of component in		
Keyed Collection Ac	Keyed Collection Acce		
🥐 Keyed Collection Ite	Keyed Collection Iterat		
A Label	Text description		
A Label	Text description		
📑 List Box	List Box is a basic list		
<ul> <li>List Box Cell</li> </ul>	List Box Cell		

- 6.22.1 Types of Collections
- 6.22.2 LANSA Collections
- 6.22.3 Keyed Collections

6.22.4 Other Optimized Collections

6.22.5 Collection Programming Summary

1 6. Creating Applications Using Components

# 6.22.1 Types of Collections

```
The type of collection will depend on its purpose, you can choose from:
Keyed Collection (PRIM_KCOL)
List Collection (PRIM_LCOL)
Array Collection (PRIM_ACOL)
Sorted Array Collection (PRIM_SACO)
Set Collection (PRIM_SCOL)
Dictionary Collection (PRIM_DCOL)
Sorted Dictionary Collection (PRIM_SDCO)
```

# **Keyed Collection (PRIM\_KCOL)**

Keyed collections are an unordered sequence of components identified by a key value. No duplicates of the key value are allowed. A keyed collection can be keyed by any field in the repository.

Define\_Com Class(#prim\_kcol<#Prim\_Form #Std\_num>) Name(#Forms) Adding items to the collection is relatively slow as the key has to be maintained. However, lookup in the collection is comparatively fast as the key is used to directly access the item.

```
#Form <= #Forms<#std_num>
```

or

```
#Form <= #Forms<123>
```

Iterating over the collection using the FOR command will return the items in key order.

Keyed collections are best suited to relatively small amounts of data used where fast lookup is required.

# List Collection (PRIM\_LCOL)

List collections provide an ordered collection of components. The features of the list component are positional in nature, in reference to a given index or to the beginning or end of the list. Indexing is always relative to 1.

```
Define_Com Class(#prim_lcol<#Prim_Form>) Name(#Forms)
```

Adding items to the collection is relatively quick as there is no key to maintain. However, lookup in the collection is relatively slow due to the sequential nature of the list unless the specific index is known. Items can be added at specific positions, for example, first, last, before or after a specific index.

```
#Forms.InsertBefore(#Form 5)
```

Iterating over the collection using the FOR command will return the items in the order that they were added.

List collections are best suited to situations where the data will be collected and then later iterated over.

# Array Collection (PRIM\_ACOL)

Array collections provide a dynamically sized, ordered collection of components. The features of the array component are positional in nature, in reference to a given index or to the beginning or end of the collection. Indexing is always relative to 1.

Define\_Com Class(#prim\_acol<#Prim\_Form>) Name(#Forms)

Adding items to the collection is relatively quick as there is no key to maintain. However, lookup in the collection is relatively slow due to the sequential nature of the list unless the specific index is known. Items can be added at specific positions, for example, first, last, before or after a specific index.

```
#Forms.InsertFirst(#Form)
```

Iterating over the collection using the FOR command will return the items in the order that they were added.

Array collections are best suited to situations where the data will be continually and directly accessed.

# Sorted Array Collection (PRIM\_SACO)

Sorted Array collections provide a dynamically sized, sorted collection of components. The features of the array component are positional in nature, in reference to a given index or to the beginning or end of the collection. Indexing is always relative to 1.

Define\_Com Class(#prim\_saco<#Prim\_Form>) Name(#Forms) Adding items to the collection is relatively slow as the collection is sorted. Whenever the collection needs to sequence two objects it fires the Compare event. This allows you to determine the sort sequence for the items based on whatever arbitrary sequence you need.

Lookup in the collection is relatively slow due to the non-keyed nature of the list unless the specific index is known.

#Forms.Insert(#Form)

Iterating over the collection using the FOR command will return the items based on the order of the sort defined.

Sorted Array collections are best suited to situations where the data will be continually and directly accessed and you want to determine the sort sequence.

# Set Collection (PRIM\_SCOL)

Set collections provide an unordered collection of components. Set collections cannot contain duplicate instances.

Define\_Com Class(#prim\_scol<#Prim\_Form>) Name(#Forms)

Adding items to the collection is relatively quick as there is no key to maintain. However, lookup in the collection is relatively slow due to the non-keyed nature of the collection.

#Forms.Insert(#Form)

Iterating over the collection using the FOR command will return the items in a seemingly random order.

Set collections are best suited to processing large collections of objects to ensure a singularity.

#### **Dictionary Collection (PRIM\_DCOL)**

Dictionary collections are an unordered sequence of key-value component pairs. Duplicate keys are not allowed.

```
Define_Com Class(#prim_dcol<#Prim_Form #Prim_objt>) Name(#Forms)
```

Adding items to the collection is relatively slow as the key has to be maintained. However, lookup in the collection is comparatively fast, as the key is used to directly access the item.

```
#Forms.Insert(#Form #Key)
```

Iterating over the collection using the FOR command will return the items in

key order. However, as the key is a component the order will appear to be random.

Dictionary collections are best suited to relatively small amounts of data where fast lookup is required.

# Sorted Dictionary Collection (PRIM\_SDCO)

Dictionary collections are a sorted sequence of key-value component pairs. Duplicate keys are not allowed.

```
Define_Com Class(#prim_sdco<#Prim_Form #Prim_objt>) Name(#Forms)
```

Adding items to the collection is relatively slow as the collection is sorted. Whenever the collection needs to sequence two objects it fires the Compare event. This allows you to determine the sort sequence for the items based on whatever arbitrary sequence you require.

```
#Forms.Insert(#Form #Key)
```

Iterating over the collection using the FOR command will return the items based on the order of the defined sort.

Sorted Dictionary collections are best suited to relatively small amounts of data where fast lookup is required and you want to determine the sort sequence.

↑ 6.22 Collections

# 6.22.2 LANSA Collections

In addition to user-defined collections the Visual LANSA environment provides you with access to collections that it maintains. For example the status bar provides a Messages collection that allows you to access all the messages currently being displayed. Lists, grids and tree views have a Columns and an Items collection which you can use to access all or individual columns and items in them. Forms and other containers have a ComponentControls collection which you can use to access all the controls on the form.

↑ 6.22 Collections

#### 6.22.3 Keyed Collections

Keyed collections are a general-purpose type of collection. You need to know how to use keyed collections to write Visual LANSA applications. Following are some examples:

Example 1: Create a Keyed Collection Example 2: Collections do not Need to Be Keyed by Index Example 3: Collection to Manage Forms Factory and Collection Style Collections Example 4: Collection of Menu Items Example 5: Dynamically Create Buttons  $\hat{T}$  6.22 Collections

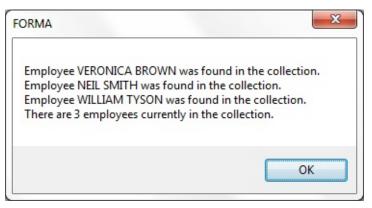
# **Example 1: Create a Keyed Collection**

This example form shows how to create a collection of employees and to add items to the collection. In this example the collection is used simply like an array.

When you type in an employee name and click on the Save button, the employee name is saved in a collection.



When you click on the Show button, all employees entered in the collection are displayed in a message box.



When the application ends, the entries in the collection are cleared. Define the Collection

Add Items to the Collection

Work with the Collection Items

Source Code for Collection Example 1

1 6.22.3 Keyed Collections

# **Define the Collection**

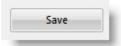
The collection to store employee names is defined like this:

Define\_Com Class(#Prim\_kCol<#FullName #Std\_Num>) Name(#Employee)

- The #FullName value indicates the collection contains #FullName fields
- The #STD\_NUM parameter specifies this repository field used as the key to the collection (a packed (7,0) number)
- The Name of the collection is #Employee.

↑ Example 1: Create a Keyed Collection

#### Add Items to the Collection



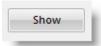
The code to Save the employee's full name into the keyed collection is contained in the Click event of the Save button:

Change Field(#LISTCOUNT) To('#Employee.ItemCount + 1') Set Com(#Employee<#ListCount>) Value(#FullName.Value)

In the above note that:

- The keyed collection property ItemCount (the number of items in the collection) is used to compute the next available key
- When you work with an item in a collection, you refer to it with the name of the collection and the key field: #Employee<#ListCount>
- You create the item in the collection simply by setting its value
- ↑ Example 1: Create a Keyed Collection

#### Work with the Collection Items



Individual items in the collection are accessed using a For/EndFor loop. The click event of the Show button iterates through the collection referencing each entry in the collection by using a For/EndFor loop to retrieve the value of #FullName of the individual employees to a message box:

For Each(#EmployeeFullName) In(#Employee) Use Builtin(MESSAGE\_BOX\_ADD) With\_Args('Employee' #EMPLOYEEFU Endfor

↑ Example 1: Create a Keyed Collection

## **Source Code for Collection Example 1**

To see how a collection is created, copy and paste this code to a form and then compile and execute the form.

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(88) Clientwidth(

\* Using a Keyed Collection (PRIM\_KCOL) as a simple array.

\* Define the keyed collection to be used to store employee names.

- \* The collection is named #Employee and it collects #FullNames.
- \* It is indexed (or more correctly, keyed by) #Std\_Num ie: a packed (7,0) num

Define\_Com Class(#Prim\_kCol<#FullName #Std\_Num>) Name(#Employee)

- \* Define the form.
- \* #FULLNAME allows employee names to be input.
- \* Button #PHBN\_SAVE saves employee names into the collection.
- \* Button #PHBN\_SHOW shows the details of all the employees in the collecti-

Define\_Com Class(#FULLNAME.Visual) Name(#FULLNAME) Displayposit Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_SAVE) Buttondefault(True Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_SHOW) Caption('Show ')

\* \_\_\_\_\_

\* #PHBN\_SAVE.Click : Save the detail on an employee in the keyed collection

\* \_\_\_\_\_

Evtroutine Handling(#PHBN\_SAVE.Click)

\* Save the employee's full name into the keyed collection in the next slot.

\* Use the keyed collection property ItemCount to compute the next slot

Change Field(#LISTCOUNT) To('#Employee.ItemCount + 1') Set Com(#Employee<#ListCount>) Value(#FullName.Value)

\* Clear the full name field ready for input of the next name

#### Change Field(#FULLNAME) To(\*NULL)

#### Endroutine

\* \_\_\_\_\_

\* #PHBN\_SHOW.Click : Build a message of all the employees saved

\* \_\_\_\_\_

Evtroutine Handling(#PHBN\_SHOW.Click) Define Field(#LOOPINDEX) Reffld(#STD\_NUM)

\* Iterate through the collection referencing each entry in the

\* collection as an object by using a for / end for loop.

For Each(#EmployeeFullName) In(#Employee) Use Builtin(MESSAGE\_BOX\_ADD) With\_Args('Employee' #EMPLOYEEFU Endfor

\* Show the employee count and final results

Use Builtin(MESSAGE\_BOX\_ADD) With\_Args('There are' #EMPLOYEE.IT Use Builtin(MESSAGE\_BOX\_SHOW) With\_Args(OK OK INFO \*COMPON

Endroutine

End\_Com

↑ Example 1: Create a Keyed Collection

## **Example 2: Collections do not Need to Be Keyed by Index**

Unlike an array, a keyed collection does not have to be keyed by an index. This example is otherwise the same as Example 1 except that the collection of employees is keyed by the #Surname field instead of a number.

When you type in an employee name and click on the Save button, the employee name is saved in a collection.

1		
Employee Given Name(s)	BEN	
Employee Surname	SMITH	
	Save	Show

When you click on the Show button, all employees entered in the collection are displayed in a message box.

FORMA	×
Employee MARY JONES was Employee BEN SMITH was fo There are 2 employees curren	ound in the collection.
	ОК

The collection is uniquely keyed by #SurName so a duplicate name will update the existing item in the collection.

Define the Collection

Add Items to the Collection

Work with the Collection Items

Source Code for Collection Example 2

1 6.22.3 Keyed Collections

## **Define the Collection**

The collection to store employee names is defined like this:

Define\_Com Class(#Prim\_kCol<#GiveName #SurName>) Name(#Employee)

- The collection is named #Employee.
- It collects an employee's #GiveName (given or first name).
- It is uniquely keyed an employee's #SurName (last name or surname).

#### Add Items to the Collection

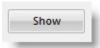


The code to save the employee's full name into the keyed collection is contained in the Click event of the Save button:

Set Com(#Employee<#SurName>) Value(#GiveName)

In the above note that there is no need to compute the next available key for the collection item as in Example 1 because the collection is keyed by the #Surname field.

#### Work with the Collection Items



Individual items in the collection are accessed using a For/EndFor loop. The click event of the Show button iterates through the collection referencing each entry in the collection by using a For/EndFor loop to retrieve the value of #GiveName and #Surname of the individual employees to a message box:

For Each(#Employee\_GiveName) In(#Employee) Key(#Employee\_SurName) Use Builtin(MESSAGE\_BOX\_ADD) With\_Args('Employee' #EMPLOYEE\_C Endfor

In the above note that:

• The For/EndFor loop specifies a name for the Key (#Employee\_SurName) to access the key of the current component each time the variable name (#Employee\_GiveName) is referenced. In this way the value of the #Surname field can be retrieved to the message.

# **Source Code for Collection Example 2**

To see how a collection is created, copy and paste this code to a form and then compile and execute the form.

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(124) Clientwidth

\* Using a Keyed Collection (PRIM\_KCOL) as a keyed array.

\* Define the keyed collection to be used to store employee names.

\* The collection is named #Employee.

\* It collects an employee's #GiveName (given or first name).

\* It is uniquely keyed an employee's #SurName (last name or surname).

Define\_Com Class(#Prim\_kCol<#GiveName #SurName>) Name(#Employee)

\* Define the form.

- \* Field #GIVENAME allows the employee's given/first name to be input.
- \* Field #SURNAME allows the employee's last or surname to be input.
- \* Button #PHBN\_SAVE saves employee names into the collection.
- \* Button #PHBN\_SHOW shows the details of all the employees in the collection

Define\_Com Class(#GiveName.Visual) Name(#GiveName) Displayposition(2) Define\_Com Class(#SurName.Visual) Name(#SurName) Displayposition(1) H Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_SAVE) Buttondefault(True Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_SHOW) Caption('Show ')

\* \_\_\_\_\_

\* #PHBN\_SAVE.Click : Save the detail on an employee in the keyed collection \* \_\_\_\_\_

Evtroutine Handling(#PHBN\_SAVE.Click)

\* Save the employee's name into the keyed collection.

\* The collection is uniquely keyed by #SurName so a duplicate

\* name will update the existing item in the collection. Try putting

\* in 2 employees with the same surname (eg: JONES).

Set Com(#Employee<#SurName>) Value(#GiveName)

\* Clear the name fields ready for input of the next name \* and reset the focus to the first name field

Change Field(#GIVENAME #SURNAME) To(\*NULL) Invoke Method(#GiveName.SetFocus)

Endroutine

\* \_\_\_\_\_

\* #PHBN\_SHOW.Click : Build a message of all the employees saved

\* \_\_\_\_\_

Evtroutine Handling(#PHBN\_SHOW.Click)

\* Iterate through the collection referencing each entry in the

\* collection as an object by using a for / end for loop.

For Each(#Employee\_GiveName) In(#Employee) Key(#Employee\_SurName) Use Builtin(MESSAGE\_BOX\_ADD) With\_Args('Employee' #EMPLOYEE\_C Endfor

\* Show the employee count and final results

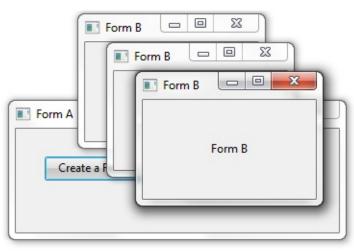
Use Builtin(MESSAGE\_BOX\_ADD) With\_Args('There are' #EMPLOYEE.IT Use Builtin(MESSAGE\_BOX\_SHOW) With\_Args(OK OK INFO \*COMPON

Endroutine

End\_Com

### **Example 3: Collection to Manage Forms**

This example shows a form which manages its child forms using a keyed collection:



When you click on the Create a Form B button, a new instance of Form B is created and added to the collection. You can use the other two buttons to move the forms right or left.

Define the Collection

Add Items to the Collection

Work with the Collection Items

Source Code for Collection Example 3

1 6.22.3 Keyed Collections

## **Define the Collection**

The collection to store child forms is defined like this:

Define\_Com Class(#Prim\_Kcol<#FormB #Std\_Num>) Name(#FormBs)

- The collection is named #FormBs and it collects components named #FormB.
- It is keyed by #Std\_Num ie: a packed (7,0) number.
- 1 Example 3: Collection to Manage Forms

#### Add Items to the Collection

Create a Form B

The code to create a new form B in the collection is contained in the Click event of the Create a Form B button:

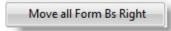
Change Field(#LISTCOUNT) To('#FormBs.ItemCount + 1') Invoke Method(#FormBs<#listcount>.showform)

In the above note that:

- The keyed collection property ItemCount is used to get the next slot in the collection
- When the form is created and added to the collection when it is first shown.
- ↑ Example 3: Collection to Manage Forms

#### Work with the Collection Items

The click events of the two buttons to move the child forms to right and left use two different ways of working with items in a collection.



The click event of the Move all Form Bs Right button invokes a MoveRight method of Form B for all forms in the collection:

```
Invoke Method(#formBs<>.MoveRight)
```

To work with all items in the collection specify a blank key for the collection: #CollectionName<>

Move all Form Bs Left

The click event of the Move all Form Bs Left uses a For/EndFor loop to iterate through the items in the collection and to move them to the left:

```
For Each(#Current) In(#FormBs)
Change Field(#NEW_LEFT) To('#Current.Left - 5')
Set Com(#Current) Left(#New_Left)
Endfor
```

#### Also see

- 6.19 Multi-Form Applications
- ↑ Example 3: Collection to Manage Forms

## Source Code for Collection Example 3

#### Form A

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Caption('Form A') Clientheigh

\* Using a Keyed Collection (PRIM\_KCOL) to handle a multiform application.

\* Define the keyed collection to be used to store the child forms.

\* The collection is named #FormBs and it collects #FormBs.

\* It is keyed by #Std\_Num ie: a packed (7,0) number.

Define\_Com Class(#Prim\_Kcol<#FormB #Std\_Num>) Name(#FormBs)

- \* Define the form.
- \* CreateFormBtn will create an instance of FormB
- \* MoveRightBtn will move all the child FormBs to the right

\* MoveLeftBtn will move all the child FormBs to the left

Define\_Com Class(#PRIM\_PHBN) Name(#CreateFormBtn) Caption('Create a Define\_Com Class(#PRIM\_PHBN) Name(#MoveRightBtn) Caption('Move all Define\_Com Class(#PRIM\_PHBN) Name(#MoveLeftBtn) Caption('Move all ]

\* Grid layout manager to position the buttons Define\_Com Class(#PRIM\_GDLM) Name(#GDLM\_1) Define\_Com Class(#PRIM\_GDLI) Name(#GDLI\_1) Manage(#CreateFormBtt Define\_Com Class(#PRIM\_GDLI) Name(#GDLI\_2) Left(2) Manage(#MoveR Define\_Com Class(#PRIM\_GDLI) Name(#GDLI\_4) Left(2) Manage(#MoveL

\* Definition of child form (FormB) Define\_Com Class(#FORMB) Name(#FORMB) Componentversion(1)

\* Fields used to position the child forms (FormB) Define Field(#NEW\_TOP) Reffld(#STD\_NUM) Define Field(#NEW\_LEFT) Reffld(#STD\_NUM)

Evtroutine Handling(#CreateFormBtn.Click) \* Use the keyed collection property ItemCount to get the next slot in the collec Change Field(#LISTCOUNT) To('#FormBs.ItemCount + 1') \* Show the form. When the form is shown, it is created and added to the collec Invoke Method(#FormBs<#listcount>.showform)

\* Adjust the form's position from the left and top Change Field(#NEW\_LEFT) To('#FORMBS<#listcount>.Left + (#listcount \* : Change Field(#NEW\_TOP) To('#FORMBS<#listcount>.top + (#listcount \* 30) Set Com(#formBs<#listcount>) Left(#NEW\_LEFT) Top(#NEW\_TOP) Endroutine

Evtroutine Handling(#MoveRightBtn.Click) \* Invoke the MoveRight method in FormB to move all forms in the collection Invoke Method(#formBs<>.MoveRight) Endroutine

Evtroutine Handling(#MoveLeftBtn.Click)

\* Use a For/EndFor loop to iterate through all the forms in the collection and to For Each(#Current) In(#FormBs) Change Field(#NEW\_LEFT) To('#Current.Left - 5') Set Com(#Current) Left(#New\_Left) Endfor Endroutine End Com

Form B

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Caption('Form B') Clientheigh \* Define the child FormB used by FormA

\* Define a label Define\_Com Class(#PRIM\_LABL) Name(#LABL\_1) Alignment(Center) Cap

\* Center the label on the form using an attachment layout Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1) Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Center) Mana

\* Define a field to position the form Define Field(#NEW\_RIGHT) Reffld(#STD\_NUM) \* Method to move the form to the right by incrementing the value of its Left pr Mthroutine Name(MoveRight) Help('This method moves the form to the right. Change Field(#NEW\_RIGHT) To('#COM\_OWNER.LEFT + 5') Set Com(#com\_owner) Left(#NEW\_RIGHT) Endroutine

End\_Com

When you create a new form and specify an ancestor for it, the inheriting form will look and behave exactly like its ancestor form.

1 Example 3: Collection to Manage Forms

# **Factory and Collection Style Collections**

There are two different ways of creating a keyed collection, one with collection style Factory and one with collection style Collection.

By default the Style property of a keyed collection is Factory. This means that a new item in the collection is created whenever the collection is accessed using a value in the key field that does not already exist in the collection.

Alternatively the Style can be Collection. This means that the application must explicitly create the items in the collection by using a SET\_REF command.

The advantage of using references is that you can have direct control over memory usage by explicitly creating and destroying components. The larger and more complex your application is, the more important it becomes to ensure that your application uses memory effectively.

# **Factory Collections**

For any given object called #Object defined in a collection like this (where the default Style() is shown):

Define\_Com Class(#Prim\_KCol<#Object #Key>) Name(#Collection) Styl

any reference to an item in the collection, for example:

Set #Collection<#Key> .....

will automatically create an instance of #Object if an object with key #Key is not already in the collection.

### **Collection Style Collections**

If the collection is defined like this:

Define\_Com Class(#Prim\_KCol<#Object #Key>) Name(#Collection) Style

then

Set #Collection<#Key> .....

will fail with an error message if an object with key #Key is not already in the

collection. To handle a Style(Collection) collection correctly in this situation you need to code this:

```
If_Ref #Collection<#Key> is(*null)
Set_Ref #Collection<#Key> To(*Create_as #Object)
Endif
```

```
Set #Collection<#Key> .....
```

This code first checks whether #Collection<#Key> exists in the collection. If it does not then a Set\_Ref command is used to create a new instance of #Object and then assign the reference to the new object into the collection.

This may seem like more code, but in fact Style(Collection) collections are very useful because you can check whether something exists in the collection and then control when and how it is added.

Note that only keyed collections support the Style(Factory) and Style(Collection) properties. All other collection types (eg: array, list, sorted array, etc) are implicitly Style(Collection) collections.

## Factory and Collection Style Example

This example shows you the difference between coding factory and collection style collections.

The factory buttons are automatically created when the properties of the buttons are set:

Set Com(#Button\_Factory<#Use\_Key>) Parent(#Com\_Owner) Height(20) Wi

The collection buttons are explicitly created using a SET\_REF command after which the properties of the buttons are set:

```
Set_Ref Com(#Button_Collection<#Use_Key>) To(*create_as #Prim_Phbn)
Set Com(#Button_Collection<#Use_Key>) Parent(#Com_Owner) Height(20) '
```

In both styles of collection the buttons are made visible by using the Realize method:

```
Invoke Method(#Button_Collection<>.Realize)
```



Source for the Factory/Collection Example

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(383) Clientwidth

\* Style(Factory) and Style(Collection) collections

\* Define a Style(Factory) keyed collection of buttons

Define\_Com Class(#Prim\_kCol<#Prim\_PHBN #Std\_Num>) Name(#Button\_F

\* Define a Style(Collection) keyed collection of buttons

Define\_Com Class(#Prim\_kCol<#Prim\_PHBN #Std\_Num>) Name(#Button\_C

\* Form layout is 2 buttons to initiate the creation of collection of buttons

Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_FACTORY) Caption('Mak Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_COLLECTION) Caption('

\* Incrmental value used to position buttons

Define Field(#NEXT\_TOP) Reffld(#STD\_NUM) Default(80)

Define Field(#USE\_KEY) Reffld(#STD\_NUM)

\* -----\* #PHBN FACTORY : Create 3 more buttons in the Style(Factory) collection

\* \_\_\_\_\_

Evtroutine Handling(#PHBN\_FACTORY.Click)

Begin\_Loop To(3)

\* Get the next key to be used

Change Field(#USE\_KEY) To('#Button\_Factory.ItemCount + 1')

\* Create the button simply by referencing it. Since the item does

\* not exist in the collection and the collection is Style(Factory)

\* then a new button will be automatically ""manufactured" .....

Set Com(#Button\_Factory<#Use\_Key>) Parent(#Com\_Owner) Height(20) Wi

\* Increment the next top position

Change Field(#NEXT\_TOP) To('#Next\_Top + 26') End\_Loop

\* Now that the buttons have been created make them visible by realizing them Invoke Method(#Button\_Factory<>.Realize)

Endroutine

```
* _____
* #PHBN_COLLECTION : Create 3 more buttons in the Style(Collection) coll
* _____
```

Evtroutine Handling(#PHBN\_COLLECTION.Click)

Begin\_Loop To(3) Change Field(#USE\_KEY) To('#Button\_Collection.ItemCount + 1')

Set\_Ref Com(#Button\_Collection<#Use\_Key>) To(\*create\_as #Prim\_Phbn)
Set Com(#Button\_Collection<#Use\_Key>) Parent(#Com\_Owner) Height(20) '

Change Field(#NEXT\_TOP) To('#Next\_Top + 26')

End\_Loop

\* Now that the buttons have been created make them visible by realizing them Invoke Method(#Button\_Collection<>.Realize)

Endroutine

End\_Com

↑ Factory and Collection Style Collections

## **Example 4: Collection of Menu Items**

This example shows how to create a collection of menu items which correspond the the selected entries in a list view.

Selected Employees			
Employee Nu	Employee Surname	Employee Given Nam	
A0070	Brown	Veronica	
A0090	Bloggs	Fred	
A0193	Smithson	Fred	
A0907	Simpson	Anne	
A1001	Jones	Shirley	
A1002	Smythe	John	
A1003	Smithe	Robert	
A1004	Smithson	Ruth	
A1005	Smiths	Peter	
A1006	Smithers	Jack	
A1007	Snell	George	
A1008	Sneddon	Alison	
A1009	Snashall	Damian	
A1010	Perry	Kate	
A1011	Perron	Christopher	
A1012	Paul	Patrick	
A1013	Pattinson	George	1

# **Define the Collection**

In this example we are using a collection-style collection to add menu items to the menu item collection. This is how the collection of the menu items is defined:

Define\_Com Class(#PRIM\_KCOL<#PRIM\_MITM #EMPNO>) Name(#MEN

- The Name of the collection is #MENU\_ITEMS.
- The collection will contain menu items (#PRIM\_MITM).
- The Keyedby property specifies the repository field (#EMPNO) used as the key to the collection.
- Its Style is Collection

#### Submenu

The submenu which will contain the menu items is defined, but it is not associated with the menu title yet because initially it contains no items and it is defined with a dynamic reference which means that the submenu will be created only when a reference is explicitly set to it:

DEFINE\_COM class(#PRIM\_SMNU) name(#Sub\_menu) reference(\*dynamic

↑ Example 4: Collection of Menu Items

## Add Items to the Collection

A menu item is added to the MENU\_ITEMS collection when an employee in the list is selected:

Evtroutine Handling(#LTVW\_1.ItemGotSelection) Options(\*NOCLEARMES \* Create submenu item if it doesn't already exist If\_Ref Com(#Sub\_menu) Is(\*null) Set\_Ref Com(#Sub\_menu) To(\*create\_as #prim\_smnu) Set Com(#Sub\_menu) Parent(#menu\_title) Endif

\* Create menu item based on the employee details Set\_Ref Com(#menu\_items<#empno>) To(\*create\_as #prim\_mitm)

\* Define menu item's text Use Builtin(BCONCAT) With\_Args(#EMPNO #GIVENAME #SURNAME) \* Set menu item's properties Set Com(#menu\_items<#empno>) Caption(#std\_textl) Parent(#Sub\_menu) Endroutine

↑ Example 4: Collection of Menu Items

### **Destroy an Item in the Collection**

When a list entry loses selection, the menu item is destroyed from memory by setting its reference to \*NULL:

EVTROUTINE handling(#LTVW\_1.ItemLostSelection) options(\*NOCLEAR] SET\_REF com(#menu\_items<#empno>) to(\*null) ENDROUTINE

↑ Example 4: Collection of Menu Items

#### **Source Code for Collection Example 4**

To see how the Collection-Style collection works, copy and paste this code to a form and then compile and execute the form.

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(310) Clientwidth Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1) Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Center) Mana

Define\_Com Class(#PRIM\_MBAR) Name(#MBAR\_1) Parent(#COM\_OWNI Define\_Com Class(#PRIM\_MITM) Name(#MENU\_TITLE) Caption('Selected

Define\_Com Class(#PRIM\_LTVW) Name(#LTVW\_1) Displayposition(1) Full Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Parei Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Displayposition(2) Parei Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Displayposition(3) Parei

\* Collection of selected menu items

\* STYLE(Collection) requires that the items in the collection are explicitly cre-Define\_Com Class(#PRIM\_KCOL<#PRIM\_MITM #EMPNO>) Name(#MEN

\* Submenu component required for menu items Define\_Com Class(#PRIM\_SMNU) Name(#Sub\_menu) Reference(\*dynamic)

\* Populate list view with known employees Evtroutine Handling(#COM\_OWNER.CreateInstance) Options(\*NOCLEARM Select Fields(#LTVW\_1) From\_File(PSLMST) Add\_Entry To\_List(#LTVW\_1) Endselect Endroutine

\* Create a menu item of selected employees Evtroutine Handling(#LTVW\_1.ItemGotSelection) Options(\*NOCLEARMES \* Create submenu item if it doesn't already exist If\_Ref Com(#Sub\_menu) Is(\*null) Set\_Ref Com(#Sub\_menu) To(\*create\_as #prim\_smnu) Set Com(#Sub\_menu) Parent(#menu\_title)

#### Endif

\* Create menu item based on the employee details Set\_Ref Com(#menu\_items<#empno>) To(\*create\_as #prim\_mitm)

\* Define menu item's text Use Builtin(BCONCAT) With\_Args(#EMPNO #GIVENAME #SURNAME) \* Set menu item's properties Set Com(#menu\_items<#empno>) Caption(#std\_textl) Parent(#Sub\_menu) Endroutine

\* When a list entry loses selection, destroy the menu item Evtroutine Handling(#LTVW\_1.ItemLostSelection) Options(\*NOCLEARMES Set\_Ref Com(#menu\_items<#empno>) To(\*null) Endroutine

\* When an employee is selected in the menu, deselect it in the list Evtroutine Handling(#menu\_items<>.Click) Com\_Sender(#Selected\_menu\_ite Define Field(#W\_EMPNO) Reffld(#EMPNO) \* Use menu item caption to find out which employee number this item represe Change Field(#STD\_TEXTL) To('#Selected\_menu\_items.Caption') Substring Field(#STD\_TEXTL 1 5) Into\_Field(#W\_EMPNO) Selectlist Named(#LTVW\_1) Continue If('#w\_empno \*ne #empno') \* Set this employee to be unselected Set Com(#ltvw\_1.currentitem) Selected(false) \* Destroy the menu item Set\_Ref Com(#menu\_items<#w\_empno>) To(\*NULL) Leave Endselect Endroutine

End\_Com

## **Example 5: Dynamically Create Buttons**

There are many programming situations where an action requires the creation and population of a collection, passing the collection to another component for processing and finally destroying the collection.

This example shows how to dynamically create a keyed collection of buttons, work with them, and how to then destroy all the buttons from memory:

Setting References Number of buttons to Create	12 ¢ Create Button	Destroy all Buttor	nable All Button	Disable All Butt
Push Button 0000001	Push Button 0000002	Push Button 000000	Butt	on 0000004
Push Button 0000005	Push Button 0000006	Push Button 000000	Push Butt	on 0000008
Push Button 0000009	Push Button 0000010	Push Button 000001	Push Butt	on 0000012

A Dynamic Collection Collection and Buttons Are Created at Run-Time Destroy the Collection Source Code for Collection Example 5 1 6.22.3 Keyed Collections

# **A Dynamic Collection**

This example first creates a dynamic collection of buttons:

DEFINE\_COM class(#PRIM\_KCOL<#prim\_phbn #STD\_NUM>) name(#BU By specifying reference(\*DYNAMIC) we are telling LANSA not to create the collection when the form is loaded (as happens with an ordinary DEFINE\_COM statement), but to create it only when a reference is explicitly made to it. The benefit of explicitly creating the collection is that later on we will be able to destroy it at any given time by setting the reference to it to \*NULL.

## **Collection and Buttons Are Created at Run-Time**

Create Button

At run-time, when the user enters the number of buttons to be created and clicks the Create Buttons button and if the collection does not already exist, it is created using the SET\_REF command:

IF\_REF com(#BUTTON\_COLLECTION) is(\*null) SET\_REF com(#BUTTON\_COLLECTION) to(\*Create\_as #PRIM\_KCOL<#J

ENDIF

And then the number of buttons specified in the #STD\_NUM field are created in the collection:

BEGIN\_LOOP to(#STD\_NUM)

CHANGE field(#BUTTINDEX) to('#ButtIndex + 1') USE builtin(BCONCAT) with\_args('Push Button' #BUTTCHAR) to\_get(#STI SET\_REF com(#Button\_collection<#ButtIndex>) to(\*CREATE\_AS #PRIM\_F SET com(#BUTTON\_COLLECTION<#ButtIndex>) PARENT(#Button\_Pane]

END\_LOOP

Lastly they are realized:

- \* Realize all the buttons just created. You can do this button by button
- \* but it is more efficient to do them all in one go by (re)realizing their parent ...

INVOKE method(#Button\_Panel.Realize)

## **Destroy the Collection**

Destroy all Buttor

All buttons in the collection are be destroyed when the user clicks on the Destroy all Buttons button which sets the reference to the entire collection to \*NULL:

EVTROUTINE handling(#DESTROY\_BUTTON.Click) SET\_REF com(#BUTTON\_COLLECTION) to(\*null) ENDROUTINE

## **Source Code for Collection Example 5**

FUNCTION options(\*DIRECT) BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) CAPTION('Setting Refere

\* Form layout and button details

DEFINE\_COM CLASS(#PRIM\_PANL) NAME(#BUTTON\_PANEL) DISPL DEFINE\_COM CLASS(#PRIM\_FWLM) NAME(#BUTTON\_PANEL\_FLOW DEFINE\_COM CLASS(#STD\_NUM.Visual) NAME(#STD\_NUM) CAPTIOI DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#CREATE\_BUTTON) CAP DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#DESTROY\_BUTTON) CA DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#ENABLE\_BUTTON) CAP DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#DISABLE\_BUTTON) CAP

\* Keep an index to the number of buttons created, also in character format so tl

DEFINE field(#BUTTINDEX) length(7) decimals(0) reffld(#DATE) edit\_code DEFINE field(#BUTTCHAR) type(\*CHAR) length(7) to\_overlay(#BUTTINE

\* Keyed collections required to track buttons and flow manager items dynamic

DEFINE\_COM class(#PRIM\_KCOL<#prim\_phbn #STD\_NUM>) name(#BU DEFINE\_COM class(#PRIM\_KCOL<#prim\_fwli #STD\_NUM>) name(#FLO

\* Handle Form Create instance

\* \_\_\_\_\_

EVTROUTINE handling(#Com\_Owner.CreateInstance) SET com(#Std\_Num) VALUE(3) ENDROUTINE

\* \_\_\_\_\_

\* Handle number of buttons to create being set to silly values

\* \_\_\_\_\_

EVTROUTINE handling(#Std\_Num.Changed)

CASE of\_field(#STD\_NUM) WHEN value\_is('< 0') SET com(#Std\_Num) VALUE(1) WHEN value\_is('> 50') SET com(#Std\_Num) VALUE(50) ENDCASE

#### ENDROUTINE

\* \_\_\_\_\_

\* Handle click of the button asking to dynamically create buttons

\* \_\_\_\_\_

EVTROUTINE handling(#CREATE\_BUTTON.Click)

\* If the required keyed collections do not currently exist then create them now

IF\_REF com(#BUTTON\_COLLECTION) is(\*null) SET\_REF com(#BUTTON\_COLLECTION) to(\*Create\_as #PRIM\_KCOL<#J SET\_REF com(#FLOWITEM\_COLLECTION) to(\*Create\_as #Prim\_KCol<# ENDIF

\* Now create the requested number of buttons and keep track \* of them in the keyed collections ....

BEGIN\_LOOP to(#STD\_NUM)

CHANGE field(#BUTTINDEX) to('#ButtIndex + 1') USE builtin(BCONCAT) with\_args('Push Button' #BUTTCHAR) to\_get(#STT SET\_REF com(#Button\_collection<#ButtIndex>) to(\*CREATE\_AS #PRIM\_F SET com(#BUTTON\_COLLECTION<#ButtIndex>) PARENT(#Button\_Pane]

\* Dynamically create a new flow layout item for the button and set up its detail \* Keep a reference to it in the flow item keyed collection.

SET\_REF com(#FlowItem\_Collection<#ButtIndex>) to(\*CREATE\_AS #PRIN SET com(#FlowItem\_Collection<#ButtIndex>) PARENT(#Button\_Panel\_Flov

END\_LOOP

\* Realize all the buttons just created. You can do this button by button

\* but it is more efficient to do them all in one go by (re)realizing their parent ...

INVOKE method(#Button\_Panel.Realize)

ENDROUTINE

\* \_\_\_\_\_

\* Handle request to destroy all buttons

\* \_\_\_\_\_

EVTROUTINE handling(#DESTROY\_BUTTON.Click)

\* Destroy the button and flow item collections which will cause\* all the referenced items in the collection to be destroyed as well

SET\_REF com(#FLOWITEM\_COLLECTION) to(\*null) SET\_REF com(#BUTTON\_COLLECTION) to(\*null)

\* Reset the button index to zero .....

CHANGE field(#BUTTINDEX) to(0)

ENDROUTINE

\* \_\_\_\_\_

\* Handle request to enable all buttons

\* \_\_\_\_\_

EVTROUTINE handling(#ENABLE\_BUTTON.Click)

\* If a button collection currently exists enable all the button in it ...

IF\_REF com(#BUTTON\_COLLECTION) is\_not(\*null) SET com(#BUTTON\_COLLECTION<>) ENABLED(True) ENDIF

ENDROUTINE

\* \_\_\_\_\_

\* Handle request to enable all buttons

EVTROUTINE handling(#DISABLE\_BUTTON.Click)

\* If a button collection currently exists disable all the button in it ...

IF\_REF com(#BUTTON\_COLLECTION) is\_not(\*null) SET com(#BUTTON\_COLLECTION<>) ENABLED(False) ENDIF

ENDROUTINE

EVTROUTINE handling(#BUTTON\_COLLECTION<>.Click) com\_sender(# USE builtin(BCONCAT) with\_args('You just clicked' #CLICKEDBUTTON.C. USE builtin(MESSAGE\_BOX\_SHOW) with\_args(OK OK INFO \*COMPON) ENDROUTINE

END\_COM

# **6.22.4 Other Optimized Collections**

Keyed collections are general purpose and will solve most commercial problems. However, there are other more specialized types of collections that offer more tuned performance and different functionality:

- Array collections are used where information is stored in a contiguous sequential order and only ever accessed by an index. In such situations they are substantially faster then keyed collections. Array collections are suitable for, for instance, fast lookups.
- List collections are used to pass around large lists and in drag-and-drop operations. List collections can be merged together. List collections are unordered and therefore inserts to them are fast.
- Sorted array collections constantly maintain sort order, also after inserts.

Array Collection Example

List Collection Example 1

List Collection Example 2

Sorted Array Collection Example

↑ 6. Creating Applications Using Components

# **Array Collection Example**

This example shows how to use an array collection to store and look up information. The aim of the example is to show in a simple context the basic techniques of using array collections. In real situations you would implement an array collection with large amounts of data for performance reasons.

Array collections are a dynamically sized, ordered collection of components that can be located by indexing. Indexing is always relative to 1.

In this example Department Code, Employee Salary and Zip Code information is stored in an array collection. When an employee is selected in the list view, this information is retrieved from the array collection.

Emplo	Employee Given	Employee Surname		Employee Number	A0070	
A0070	Veronica	Brown		Employee Number	A0070	
A0090	Fred	Bloggs		Employee Surname	Brown	
A0193	Fred	Smithson				
A0907	Anne	Simpson		Employee Given Name(s)	Veronica	
A1001	Shirley	Jones		Department Code Employee Salary	ADM	
A1002	John	Smythe			ADM	
A1003	Robert	Smithe			30,000.00	
A1004	Ruth	Smithson	11	Employee Sulary	50,000.00	
A1005	Peter	Smiths		Post / Zip Code	2153	
A1006	Jack	Smithers				
A1007	George	Snell				
A1008	Alison	Sneddon				
A1009	Damian	Snashall				
A1010	Kate	Perry				
A1011	Christopher	Perron				
A1012	Patrick	Paul				
A1013	George	Pattinson *	r			

Define the Collections Add Items to the Collections Retrieve the Values from the Collections Source for the Example

↑ 6.22.4 Other Optimized Collections

## **Define the Collections**

The tree array collections used to keep collections of departments, salaries and post/zip codes are defined like this:

Define\_Com Class(#Prim\_ACol<#Deptment>) Name(#Department\_Array) Define\_Com Class(#Prim\_ACol<#Salary>) Name(#Salary\_Array) Define\_Com Class(#Prim\_ACol<#PostCode>) Name(#PostCode\_Array)

The collection definitions specify the type of object to be collected and the name of the collection.

#### Add Items to the Collections

In order to add items to the collection, you need to first create the instances of the objects to be inserted:

Define\_Com Class(#Deptment) Name(#Department\_Item) Define\_Com Class(#Salary) Name(#Salary\_Item) Define\_Com Class(#PostCode) Name(#PostCode\_Item)

After the objects have been created, their value is set and they are added to the collections using the Insert method:

Set Com(#Department\_Item) Value(#Deptment) Invoke Method(#Department\_Array.Insert) Item(#Department\_Item)

Set Com(#Salary\_Item) Value(#Salary) Invoke Method(#Salary\_Array.Insert) Item(#Salary\_Item)

Set Com(#PostCode\_Item) Value(#PostCode)
Invoke Method(#PostCode\_Array.Insert) Item(#PostCode\_Item)

#### **Retrieve the Values from the Collections**

The values stored for an employee in the collections are retrieved when an employee is selected in the list. The array collections are in the same order as the entries in the list view so we can use the list view entry number as the index to get the values from the array collections:

Change Field(#USE\_INDEX) To('#EMP\_LIST.CURRENTITEM.ENTRY')

The values in the array collection are then assigned to the Department, Salary and PostCode fields:

Set Com(#Deptment) Value(#Department\_Array.Item<#Use\_Index>.Value) Set Com(#Salary) Value(#Salary\_Array.Item<#Use\_Index>.Value) Set Com(#PostCode) Value(#PostCode\_Array.Item<#Use\_Index>.Value)

#### Source for the Example

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(298) Clientwidth

\* \_\_\_\_\_

\* This example demonstrates the use of array collections (#PRIM\_ACOL).

\* Generally array collections are used where information is stored

\* in a contiguous sequential order and only ever accessed by an index.

\* In such situations they are substantially faster then keyed collections.

\* The list view on the left containing the employee number (#EMPNO), \* the given name (#GIVENAME) and the surname (#SURNAME).

Define\_Com Class(#PRIM\_LTVW) Name(#emp\_list) Componentversion(1) E Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Parei Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Displayposition(2) Parei Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Displayposition(3) Parei

\* The fields on right hand side of the panel

Define\_Com Class(#EMPNO.Visual) Name(#EMPNO) Displayposition(2) He Define\_Com Class(#SURNAME.Visual) Name(#SURNAME) Displaypositior Define\_Com Class(#GIVENAME.Visual) Name(#GIVENAME) Displaypositi Define\_Com Class(#DEPTMENT.Visual) Name(#DEPTMENT) Displaypositi Define\_Com Class(#SALARY.Visual) Name(#SALARY) Displayposition(6) I Define\_Com Class(#POSTCODE.Visual) Name(#POSTCODE) Displaypositic

\* Define the array collections used to keep collections of\* departments, salaries and post/zip codes.

Define\_Com Class(#Prim\_ACol<#Deptment>) Name(#Department\_Array) Define\_Com Class(#Prim\_ACol<#Salary>) Name(#Salary\_Array) Define\_Com Class(#Prim\_ACol<#PostCode>) Name(#PostCode\_Array)

\* \_\_\_\_\_

\* At initialization fill the list view and array collections with employee details

\* \_\_\_\_\_

<sup>\*</sup> \_\_\_\_\_

Evtroutine Handling(#COM\_OWNER.Initialize)

Select Fields(#EMPNO #GIVENAME #SURNAME #SALARY #POSTCODE Invoke Method(#Com\_Owner.AddEmployee) Endselect

Endroutine

\* \_\_\_\_\_

\* This method routine add details of an employee to the list view and also

\* stores salary, department and post/zip code details in the array collections

\* \_\_\_\_\_

Mthroutine Name(AddEmployee)

\* Cause new instances of a department, salary and post/zip code object to be cr

Define\_Com Class(#Deptment) Name(#Department\_Item) Define\_Com Class(#Salary) Name(#Salary\_Item) Define\_Com Class(#PostCode) Name(#PostCode\_Item)

\* Add an entry to the list view

Add\_Entry To\_List(#EMP\_LIST)

\* Set up the Department, Salary and Postcode item objects and

\* then add references to them to the associated array collections.

\* We need to do this because array collections (PRIM\_ACOL) only

\* store references to objects.

Set Com(#Department\_Item) Value(#Deptment)
Invoke Method(#Department\_Array.Insert) Item(#Department\_Item)

Set Com(#Salary\_Item) Value(#Salary) Invoke Method(#Salary\_Array.Insert) Item(#Salary\_Item)

Set Com(#PostCode\_Item) Value(#PostCode)
Invoke Method(#PostCode\_Array.Insert) Item(#PostCode\_Item)

#### Endroutine

\* \_\_\_\_\_

\* Handle the selection of an item in the list view by displaying the

\* associated employee details in the fields on the right of the panel

\* \_\_\_\_\_

Evtroutine Handling(#Emp\_List.ItemGotSelection)

Define Field(#USE\_INDEX) Reffld(#STD\_NUM)

\* The values for #EMPNO, #GIVENAME and #SURNAME have been set fro.
\* the fields in the list view ..... however the #DEPARTMENT, #SALARY
\* and #POSTCODE values need to be retreived from the array collections
\* that stored them. Since the array collections are in the same order
\* as the entries in the list view we can use the list view entry number
\* as the index to get the values from the array collections .....

Change Field(#USE\_INDEX) To('#EMP\_LIST.CURRENTITEM.ENTRY')

\* Now set the fields on the right from values in the respective arrays

Set Com(#Deptment) Value(#Department\_Array.Item<#Use\_Index>.Value) Set Com(#Salary) Value(#Salary\_Array.Item<#Use\_Index>.Value) Set Com(#PostCode) Value(#PostCode\_Array.Item<#Use\_Index>.Value)

Endroutine

End\_Com

## List Collection Example 1

This example shows how to use a list collection to drag and drop items from one form to another.

List collections provide an ordered collection of components. List collection components are positional in reference to a given index or to the beginning or end of the list. Indexing is always relative to 1.

In this example employee number and salary information for selected employees is stored in a list collection at the start of a drag-and-drop operation and at the end of the operation this information is retrieved from the collection to the other form.

Employee Num	Employee Salary		Employee Niver
40070	50,125.00		Employee Num
40090	20,045.91		A1005
40193	35,000.04		
40907	34,213.04		
41001	2,345.82		
41002	25,000.04		
41003	31,000.04	A1004	
41004	21,000.04	A1005	046,700.04
41005	46,700.04		~
41006	25,000.04		
41007	26,780.04		
41008	450,000.04		
41009	31,000.04		
41010	60,000.04		
41011	25,000.04		
41012	26,456.04		
1012	79 077 04	·	

Define the Collection Add Items to the Collection Retrieve Items From the Collection Source Code for List Collection Example 1 ↑ 6.22.4 Other Optimized Collections

## **Define the Collection**

The list collection is defined dynamically in the StartDrag event of the list of employees. In this way the scope of the list collection is limited to the drag operation. The #Employee\_Number variable is created to store employee numbers of selected employees. Because the collection is dynamic, it has to be explicitly created with a SET\_REF command:

Define\_Com Class(#Prim\_LCOL<#Prim\_ALPH>) Name(#PayLoadList) Refe Define\_Com Class(#Prim\_ALPH) Name(#EmployeeNumber) Reference(\*Dyr

Set\_Ref Com(#PayLoadList) To(\*Create\_as #Prim\_LCOL<#Prim\_ALPH>)

↑ List Collection Example 1

### Add Items to the Collection

Selected employees are added to the collection in a SELECTLIST loop. The #Employee\_Number variable is assigned the employee number of selected employees and then used to identify the item added in the INSERT method to the collection. The list collection is then assigned as the payload or the drag-and-drop operation:

Selectlist Named(#DRAG\_LIST) Continue If('#Drag\_List.Currentitem.Selected \*ne True') Set\_Ref Com(#EmployeeNumber) To(\*Create\_as #Prim\_ALPH) Set Com(#EmployeeNumber) Value(#EmpNo) Invoke Method(#PayLoadList.Insert) Item(#EmployeeNumber) Endselect

Set\_Ref Com(#PayLoad) To(#PayLoadList)

↑ List Collection Example 1

#### **Retrieve Items From the Collection**

When the selected employees are dragged to the other form, the DragOver event of the list on this form is used to test that the payload to be dropped is a list collection of alphas:

Evtroutine Handling(#Drop\_List.DragOver) Acceptdrop(#AcceptDrop) Payloa If\_Ref Com(#PayLoad) Is(\*Instance\_of #Prim\_LCol<#Prim\_alph>) Set Com(#AcceptDrop) Value(True) Else Set Com(#AcceptDrop) Value(False) Endif Endroutine

Another list collection is defined in the DragDrop event of the list and then the payload is cast to the collection:

Define\_Com Class(#Prim\_LCOL<#Prim\_ALPH>) Name(#PayLoadList) Refe \* Cast the payload as a list colection of alphas ....

Set\_Ref Com(#PayLoadList) To(\*Dynamic #PayLoad)

The employees are retrieved from the collection to the list in a FOR/ENDFOR loop:

For Each(#EmployeeNumber) In(#PayLoadList) Change Field(#EMPNO) To('#EMPLOYEENUMBER.VALUE') Add\_Entry To\_List(#DROP\_LIST) Endfor

1 List Collection Example 1

### **Source Code for List Collection Example 1**

#### Source for the first form

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(319) Clientwidth

\* The Drag List Definition. Note the use of the DragStyle(Automatic) property

Define\_Com Class(#PRIM\_LTVW) Name(#DRAG\_LIST) Displayposition(1) Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Paren Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Displayposition(2) Paren

\* Define the drop form (change this name to the name you give to the second f

Define\_Com Class(#LCOL00004)

\* \_\_\_\_\_

\* Handle main form initialization

\* \_\_\_\_\_

Evtroutine Handling(#com\_owner.Initialize)

\* Fill the drag list with employee details

Select Fields(#DRAG\_LIST) From\_File(PSLMST) Add\_Entry To\_List(#DRAG\_LIST) Endselect

\* Show the drop list form (change #LCOL00004 to the name you give to the se

Invoke Method(#LCOL00004.ShowForm)

Endroutine

\* Handle starting the drag operation

\* \_\_\_\_\_

Evtroutine Handling(#Drag\_List.StartDrag) Draglist(#DragList) Continue(#Cc Define\_Com Class(#Prim\_LCOL<#Prim\_ALPH>) Name(#PayLoadList) Refe Define\_Com Class(#Prim\_ALPH) Name(#EmployeeNumber) Reference(\*Dyr

\* Dynamically create the list collection that is to to be the "payload"

\* for this drag operation

Set\_Ref Com(#PayLoadList) To(\*Create\_as #Prim\_LCOL<#Prim\_ALPH>)

\* Now fill the list collection with the employee numbers of all \* the currently selected employees

Selectlist Named(#DRAG\_LIST) Continue If('#Drag\_List.Currentitem.Selected \*ne True') Set\_Ref Com(#EmployeeNumber) To(\*Create\_as #Prim\_ALPH) Set Com(#EmployeeNumber) Value(#EmpNo) Invoke Method(#PayLoadList.Insert) Item(#EmployeeNumber) Endselect

\* Finished

Set Com(#Continue) Value(True) Set\_Ref Com(#PayLoad) To(#PayLoadList) Set Com(#DragList) Dragliststyle(Selection)

Endroutine

End\_Com

#### Source for the second form

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(310) Clientwidth

\* The Drop List Definition

Define\_Com Class(#PRIM\_LTVW) Name(#DROP\_LIST) Displayposition(1) Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Parer \* \_\_\_\_\_

\* Handle Drag Over the drop list

\* \_\_\_\_\_

Evtroutine Handling(#Drop\_List.DragOver) Acceptdrop(#AcceptDrop) Payloa

\* If the payload to be dropped is a list collection of alphas \* accept it otherwise reject it .....

If\_Ref Com(#PayLoad) Is(\*Instance\_of #Prim\_LCol<#Prim\_alph>) Set Com(#AcceptDrop) Value(True) Else Set Com(#AcceptDrop) Value(False) Endif

Endroutine

```
* _____
```

\* Handle the actual drop into the drop list

\* \_\_\_\_\_

Evtroutine Handling(#Drop\_List.DragDrop) Payload(#PayLoad) Define\_Com Class(#Prim\_LCOL<#Prim\_ALPH>) Name(#PayLoadList) Refe

\* Cast the payload as a list colelction of alphas ....

Set\_Ref Com(#PayLoadList) To(\*Dynamic #PayLoad)

\* Clear the drop list and fill it will all the employee numbers in the payload ...

Clr\_List Named(#DROP\_LIST)

For Each(#EmployeeNumber) In(#PayLoadList) Change Field(#EMPNO) To('#EMPLOYEENUMBER.VALUE') Add\_Entry To\_List(#DROP\_LIST) Endfor

Endroutine

End\_Com

1 List Collection Example 1

## **List Collection Example 2**

This example is otherwise the same as the List Collection Example 1 except that the drag-and-drop operation involves more employee details and these details are stored in an employee object:

Employee	Employee Sal	Employee	Sur Em				
A0070	50,125.00	Brown	Ver	Employ Em	ploye Employee	Given Na	Post / Zip.
A0090	20,045.91	Bloggs	Fre				
A0193	35,000.04	Smithson	Fre				
A0907	34,213.04	Simpson	An				
A1001	2,345.82	Jones	Shi				
A1002	25,000.04	Smythe	Joh				
A1003	31,000.04	Smithe	Ro				
A1004	21,000.04 A10	0§mithson	46 700 84	Smiths @	Peter	ADM	2147
A1005	46,700.04 A10	Smiths	25,000.04	Smithe	Jack	TRVL	2050
A1006	25,000.04 A10	Smithers	26.780.04	Snell	George	AUD	2164
A1007	26,780.04	Snell	Ge		9-		
A1008	450,000.04	Sneddon	Ali				
A1009	31,000.04	Snashall	Da				
A1010	60,000.04	Perry	Kat				
A1011	25,000.04	Perron	Ch				
A1012	26,456.04	Paul	Pat				
A1012	79 077 04	Dattincon	Ga				

The Employee Object Define the Collection Add Items to the Collection Retrieve Items from the Collection Source Code for List Collection Example 2 Î 6.22.4 Other Optimized Collections

## The Employee Object

The employee object is used to store employee details. This object is then used in the collection for the drag and drop operation.

The object defines and publishes these employee properties:

- Empno
- GiveName
- SurName
- Deptment
- Salary
- PostCode

↑ List Collection Example 2

### **Define the Collection**

A list collection is created dynamically to be the payload for this drag operation. It is a list collection of #Employee objects. A #DragEmployee variable is created to store the employee objects to be inserted in the collection. Because the collection is dynamic, a reference is set to it using the SET\_REF command:

Define\_Com Class(#Prim\_LCOL<#Employee>) Name(#PayLoadList) Reference Define\_Com Class(#Employee) Name(#DragEmployee) Reference(\*Dynamic Set\_Ref Com(#PayLoadList) To(\*Create\_as #Prim\_LCOL<#Employee>)

1 List Collection Example 2

### Add Items to the Collection

A Selectlist loop is used to add the selected employee objects to the list collection. The list collection is then set as the payload for the drag operation:

Selectlist Named(#DRAG\_LIST) Continue If('#Drag\_List.Currentitem.Selected \*ne True') Set\_Ref Com(#DragEmployee) To(\*Create\_as #Employee) Set Com(#DragEmployee) Pempno(#Empno) Pgivename(#GiveName) Psurna: Invoke Method(#PayLoadList.Insert) Item(#DragEmployee) Endselect

Set\_Ref Com(#PayLoad) To(#PayLoadList)

↑ List Collection Example 2

### **Retrieve Items from the Collection**

In the DragDrop event of the list on the second form a list collection is defined and the payload containing the collection of selected employees is cast to it.

```
Define_Com Class(#Prim_LCOL<#Employee>) Name(#EmployeePayLoad) F
Set_Ref Com(#EmployeePayLoad) To(*Dynamic #PayLoad)
```

The list collection is iterated through in a For/EndFor loop to retrieve the employee details:

For Each(#EmployeeObject) In(#EmployeePayLoad) Change Field(#EMPNO) To('#EMPLOYEEOBJECT.PEMPNO') Change Field(#GIVENAME) To('#EMPLOYEEOBJECT.PGIVENAME') Change Field(#SURNAME) To('#EMPLOYEEOBJECT.PSURNAME') Change Field(#POSTCODE) To('#EMPLOYEEOBJECT.PPOSTCODE') Add\_Entry To\_List(#DROP\_LIST) Endfor

↑ List Collection Example 2

### Source Code for List Collection Example 2

## **Employee Object**

Create the object as a reusable part. In this example it is named #employee. Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

\* \_\_\_\_\_ \* This is an "Employee" object

\* \_\_\_\_\_

\* Define the member variables that define an "Employee" object ....

Define\_Com Class(#Empno) Define\_Com Class(#GiveName) Define\_Com Class(#SurName) Define\_Com Class(#Deptment) Define\_Com Class(#Salary) Define\_Com Class(#PostCode)

\* Publish the properties that an "Employee" exposes .....

Define\_Pty Name(pEmpno) Get(\*Auto #Empno) Set(\*Auto #Empno) Define\_Pty Name(pGiveName) Get(\*Auto #GiveName) Set(\*Auto #GiveNam Define\_Pty Name(pSurName) Get(\*Auto #SurName) Set(\*Auto #SurName) Define\_Pty Name(pDepartment) Get(\*Auto #Deptment) Set(\*Auto #Deptment) Define\_Pty Name(pSalary) Get(\*Auto #Salary) Set(\*Auto #Salary) Define\_Pty Name(pPostCode) Get(\*Auto #PostCode) Set(\*Auto #PostCode)

 $End\_Com$ 

#### **First Form**

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(319) Clientwidth

\* The Drag List Definition. Note the use of the DragStyle(Automatic) property

Define\_Com Class(#PRIM\_LTVW) Name(#DRAG\_LIST) Displayposition(1) Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Paren Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Displayposition(2) Paren Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Displayposition(3) Paren Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_4) Displayposition(4) Paren Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_5) Displayposition(5) Paren Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_6) Displayposition(6) Paren

\* Define the drop form (change #LCOL00004 to the name you give to the secc

Define\_Com Class(#LCOL00004)

\* \_\_\_\_\_

\* Handle main form initialization

\* \_\_\_\_\_

Evtroutine Handling(#com\_owner.Initialize)

\* Fill the drag list with employee details

Select Fields(#DRAG\_LIST) From\_File(PSLMST) Add\_Entry To\_List(#DRAG\_LIST) Endselect

\* Show the drop list form (change #LCOL00004 to the name you give to the se

Invoke Method(#LCOL00004.ShowForm)

Endroutine

\* \_\_\_\_\_

\* Handle starting the drag operation

\* \_\_\_\_\_

Evtroutine Handling(#Drag\_List.StartDrag) Draglist(#DragList) Continue(#Cc Define\_Com Class(#Prim\_LCOL<#Employee>) Name(#PayLoadList) Reference Define\_Com Class(#Employee) Name(#DragEmployee) Reference(\*Dynamic)

\* Dynamically create the list collection that is to to be the "payload"

\* for this drag operation. It is a list collection of #Employee objects

Set\_Ref Com(#PayLoadList) To(\*Create\_as #Prim\_LCOL<#Employee>)

\* Now fill the list collection with the employees that are selected

Selectlist Named(#DRAG\_LIST) Continue If('#Drag\_List.Currentitem.Selected \*ne True') Set\_Ref Com(#DragEmployee) To(\*Create\_as #Employee) Set Com(#DragEmployee) Pempno(#Empno) Pgivename(#GiveName) Psurna Invoke Method(#PayLoadList.Insert) Item(#DragEmployee) Endselect

\* Finished. Set the correct return values

Set Com(#Continue) Value(True) Set\_Ref Com(#PayLoad) To(#PayLoadList) Set Com(#DragList) Dragliststyle(Selection)

Endroutine

End\_Com

#### **Second Form**

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(433) Clientwidth

\* The Drop List Definition

Define\_Com Class(#PRIM\_LTVW) Name(#DROP\_LIST) Displayposition(1) Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Displayposition(2) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_4) Displayposition(3) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_6) Displayposition(4) Parer

\* \_\_\_\_\_

\* Handle Drag Over the drop list

\* \_\_\_\_\_

Evtroutine Handling(#Drop\_List.DragOver) Acceptdrop(#AcceptDrop) Payloa

\* If the payload to be dropped is a list collection of employees \* then accept it otherwise reject it .....

If\_Ref Com(#PayLoad) Is(\*Instance\_of #Prim\_LCol<#Employee>) Set Com(#AcceptDrop) Value(True) Else Set Com(#AcceptDrop) Value(False) Endif

Endroutine

\* \_\_\_\_\_

\* Handle the actual drop into the drop list

\* \_\_\_\_\_

Evtroutine Handling(#Drop\_List.DragDrop) Payload(#PayLoad) Define\_Com Class(#Prim\_LCOL<#Employee>) Name(#EmployeePayLoad) F

\* Cast the payload as a list collection of employee objects ....

Set\_Ref Com(#EmployeePayLoad) To(\*Dynamic #PayLoad)

\* Fill drop list with all the employee numbers in the payload ...

Clr\_List Named(#DROP\_LIST)

For Each(#EmployeeObject) In(#EmployeePayLoad) Change Field(#EMPNO) To('#EMPLOYEEOBJECT.PEMPNO') Change Field(#GIVENAME) To('#EMPLOYEEOBJECT.PGIVENAME') Change Field(#SURNAME) To('#EMPLOYEEOBJECT.PSURNAME') Change Field(#POSTCODE) To('#EMPLOYEEOBJECT.PPOSTCODE') Add\_Entry To\_List(#DROP\_LIST) Endfor

Endroutine

End\_Com the contract of the second seco

## **Sorted Array Collection Example**

Sorted array collections are a dynamically sized, sorted collection of components that can be located by indexing. Indexing is always relative to 1.

In this example a form invokes a reusable part that returns a an array collection that has been sorted into the order you requested. The example also uses an employee reusable part which is used to store employee details.

Type in the name of the field on which you want the list sorted and then click the "Get List" Push button to display the sorted list. The fields you can sort by are EMPNO, SURNAME, GIVENAME, POSTCODE, SALARY, and DEPTMENT.

et List Sorted	d by Field Name SALAR	Y Get List	t			
<b>F</b>						
Employee	Employee Surname	Employee Given Na		Employee Salary	Depar ADM	-
A1509	Redford	Robert	2060	10.00		1
A1001	Jones	Shirley	2001	2,345.82	ADM	
A1404	Black	Gillian	2090	12,345.04	ADM	
A1024	Taylor	John	2002	15,000.04	MKT	
A0090	Bloggs	Fred	2220	20,045.91	FLT	
A1004	Smithson	Ruth	2144	21,000.04	ADM	
A1023	Reid	David	2001	21,000.04	LEG	
A1016	Turner	Jack	2100	22,000.04	FLT	
A1002	Smythe	John	2100	25,000.04	ADM	
A1006	Smithers	Jack	2050	25,000.04	TRVL	
A1011	Perron	Christopher	2003	25,000.04	AUD	
A1022	Thompson	Kelly	2144	25,000.29	MKT	
A1017	Neave	Gary	2016	25,600.04	INF	
A1018	Zacharia	Paul	2021	25,900.04	GAC	
A1012	Paul	Patrick	2147	26,456.04	ADM	

Define the Collection and Adding Items to It Retrieve the Items in the Collection Source for Sorted Array Collection Example ↑ 6.22.4 Other Optimized Collections

### **Define the Collection and Adding Items to It**

A sorted array collection is defined In the reusable part used to sort the employees. Employee details are retrieved from file PSLMST, #SortedEmployee objects are created based on the #Employee object and assigned the retrieved employee details. The collection is then filled with the #SortedEmployee objects:

Set\_Ref Com(#SortedEmployeeList) To(\*Create\_as #Prim\_SACO<#Employee

Select Fields(#EMPNO #GIVENAME #SURNAME #DEPTMENT #SALAR Set\_Ref Com(#SortedEmployee) To(\*Create\_as #Employee) Set Com(#SortedEmployee) Pempno(#Empno) Pgivename(#GiveName) Psurn Invoke Method(#SortedEmployeeList.Insert) Item(#SortedEmployee) Endselect

As every item is added to the collection, the Compare event is triggered. This event is used to provide the logic for sorting the collection. #ObjectEmployee and #SubjectEmployee objects are created to be used as the Object and #Subject parameters to compare the employees:

Evtroutine Handling(#SortedEmployeeList.Compare) Object(#Object) Subject Define\_Com Class(#Employee) Name(#ObjectEmployee) Reference(\*Dynami Define\_Com Class(#Employee) Name(#SubjectEmployee) Reference(\*Dynam

\* Cast the object and subject objects to the Employee object

Set\_Ref Com(#ObjectEmployee) To(\*Dynamic #Object)
Set\_Ref Com(#SubjectEmployee) To(\*Dynamic #Subject)

\* Now compare the objects based on the current #R\_OrderBy value

Case Of\_Field(#R\_ORDERBY)

When Value\_Is('= GiveName') Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pGiv

When Value\_Is('= Surname') Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pSur] When Value\_Is('= Deptment') Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pDep

When Value\_Is('= Salary') Invoke Method(#Com\_Owner.CompareNumeric) Object(#ObjectEmployee.pS

When Value\_Is('= PostCode') Invoke Method(#Com\_Owner.CompareNumeric) Object(#ObjectEmployee.pP

Otherwise Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pEmj

Endcase

Endroutine

↑ Sorted Array Collection Example

#### **Retrieve the Items in the Collection**

The form retrieves the values in the sorted collection by defining a reference to the employee list and then invoking the method in the reusable part to get the employees.

Define\_Com Class(#Prim\_SACO<#Employee>) Name(#ReturnedList) Refere

Invoke Method(#LCOL00002.GetEmployees) Employeelist(#ReturnedList) O

The collection is iterated through to add the entries to the visible #Emp\_List:

For Each(#EmployeeObject) In(#ReturnedList)

Change Field(#EMPNO) To('#EMPLOYEEOBJECT.PEMPNO') Change Field(#GIVENAME) To('#EMPLOYEEOBJECT.PGIVENAME') Change Field(#SURNAME) To('#EMPLOYEEOBJECT.PSURNAME') Change Field(#POSTCODE) To('#EMPLOYEEOBJECT.PPOSTCODE') Change Field(#DEPTMENT) To('#EMPLOYEEOBJECT.PDEPARTMENT') Change Field(#SALARY) To('#EMPLOYEEOBJECT.PSALARY')

Add\_Entry To\_List(#EMP\_LIST)

Endfor

↑ Sorted Array Collection Example

## Source for Sorted Array Collection Example

## **Employee Object**

Create this object as a reusable part. In this example it is named #Employee. Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

\* \_\_\_\_\_ \* This is an "Employee" object

\* \_\_\_\_\_

\* Define the member variables that define an "Employee" object ....

Define\_Com Class(#Empno) Define\_Com Class(#GiveName) Define\_Com Class(#SurName) Define\_Com Class(#Deptment) Define\_Com Class(#Salary) Define\_Com Class(#PostCode)

\* Publish the properties that an "Employee" exposes .....

Define\_Pty Name(pEmpno) Get(\*Auto #Empno) Set(\*Auto #Empno) Define\_Pty Name(pGiveName) Get(\*Auto #GiveName) Set(\*Auto #GiveNam Define\_Pty Name(pSurName) Get(\*Auto #SurName) Set(\*Auto #SurName) Define\_Pty Name(pDepartment) Get(\*Auto #Deptment) Set(\*Auto #Deptment) Define\_Pty Name(pSalary) Get(\*Auto #Salary) Set(\*Auto #Salary) Define\_Pty Name(pPostCode) Get(\*Auto #PostCode) Set(\*Auto #PostCode)

 $End\_Com$ 

#### **Reusable Part For Sorting The List**

In this example this component is named #LCOL00002.

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_OBJT)

Define\_Com Class(#Prim\_SACO<#Employee>) Name(#SortedEmployeeList)

Define Field(#R\_ORDERBY) Reffld(#STD\_TEXTS)

\* This method is passed an OrderBy() parameter set to EMPNO, SURNAME, \* SALARY, DEPTMENT or ZIPCODE. It extracts a list of all employees in th \* table and returns their details as a sorted array collection of employee objects

Mthroutine Name(GetEmployees) Define\_Map For(\*Output) Class(#Prim\_SACO<#Employee>) Name(#Employ Define\_Map For(\*input) Class(#Std\_Texts) Name(#OrderBy)

\* Define an Employee object reference

Define\_Com Class(#Employee) Name(#SortedEmployee) Reference(\*Dynami

\* Make the requested order by parameter visible to the #SortEmployeeList.Cor

Change Field(#R\_ORDERBY) To('#ORDERBY.VALUE')

\* Dynamically create a new sorted array list to contain the employee objects

Set\_Ref Com(#SortedEmployeeList) To(\*Create\_as #Prim\_SACO<#Employee

\* Fill the sorted array collection list with employee objects

Select Fields(#EMPNO #GIVENAME #SURNAME #DEPTMENT #SALAR)

Set\_Ref Com(#SortedEmployee) To(\*Create\_as #Employee)

Set Com(#SortedEmployee) Pempno(#Empno) Pgivename(#GiveName) Psurn

Invoke Method(#SortedEmployeeList.Insert) Item(#SortedEmployee)

Endselect

\* Return a refernence to the list that was created

Set\_Ref Com(#EmployeeList) To(#SortedEmployeeList)

\* Drop the reference in this component as we have finished with the list

Set\_Ref Com(#SortedEmployeeList) To(\*null)

Endroutine

\* This routine is invoked to compare the employees in #SortEmployeeList as tl \* by the GetEmployees method above.

Evtroutine Handling(#SortedEmployeeList.Compare) Object(#Object) Subject Define\_Com Class(#Employee) Name(#ObjectEmployee) Reference(\*Dynami Define\_Com Class(#Employee) Name(#SubjectEmployee) Reference(\*Dynam

\* Cast the object and subject objects to the Employee object

Set\_Ref Com(#ObjectEmployee) To(\*Dynamic #Object)
Set\_Ref Com(#SubjectEmployee) To(\*Dynamic #Subject)

\* Now compare the objects based on the current #R\_OrderBy value

Case Of\_Field(#R\_ORDERBY)

When Value\_Is('= GiveName') Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pGiv

When Value\_Is('= Surname') Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pSur]

When Value\_Is('= Deptment') Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pDep

When Value\_Is('= Salary') Invoke Method(#Com\_Owner.CompareNumeric) Object(#ObjectEmployee.pS

When Value\_Is('= PostCode') Invoke Method(#Com\_Owner.CompareNumeric) Object(#ObjectEmployee.pP

Otherwise Invoke Method(#Com\_Owner.CompareAlpha) Object(#ObjectEmployee.pEmj Endcase

Endroutine

\* Alpha compare routine

Mthroutine Name(CompareAlpha) Define\_Map For(\*input) Class(#Std\_TextL) Name(#Object) Define\_Map For(\*input) Class(#Std\_TextL) Name(#Subject) Define\_Map For(\*output) Class(#Std\_Texts) Name(#Result)

If Cond('#Subject.Value < #Object.Value') Set Com(#Result) Value(Less) Else If Cond('#Subject.Value > #Object.Value') Set Com(#Result) Value(Greater) Else Set Com(#Result) Value(Equal) Endif Endif

Endroutine

\* Numeric compare routine

Mthroutine Name(CompareNumeric) Define\_Map For(\*input) Class(#dem\_dptot) Name(#Object) Define\_Map For(\*input) Class(#dem\_dptot) Name(#Subject) Define\_Map For(\*output) Class(#Std\_Texts) Name(#Result)

If Cond('#Subject.Value < #Object.Value') Set Com(#Result) Value(Less) Else If Cond('#Subject.Value > #Object.Value') Set Com(#Result) Value(Greater) Else Set Com(#Result) Value(Equal) Endif Endif

Endroutine

End\_Com

#### The Form

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(330) Clientwidth

\* Define the component that returns the sorted employee number list

Define\_Com Class(#PRIM\_LTVW) Name(#Emp\_LIst) Componentversion(1) Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Displayposition(1) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Displayposition(2) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_3) Displayposition(3) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_4) Displayposition(4) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_5) Displayposition(5) Parer Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_6) Displayposition(6) Parer Define\_Com Class(#STD\_OBJ.Visual) Name(#Sort\_By\_Field) Caption('Get L Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_Get\_List) Caption('Get List)

\* Define the component that builds the sorted employee lists

\* (in this example it is called #LCOL00002, change it to the name you have

\* given to the reusable part)

Define\_Com Class(#LCOL00002)

\* Handle form initialization

Evtroutine Handling(#com\_owner.Initialize)

Set Com(#Sort\_By\_Field) Value(SALARY)

Endroutine

Evtroutine Handling(#PHBN\_Get\_List.Click)

\* Define a reference to the employee list that will be returned

Define\_Com Class(#Prim\_SACO<#Employee>) Name(#ReturnedList) Refere

\* Invoke the method that returns the sorted list of employees

Invoke Method(#LCOL00002.GetEmployees) Employeelist(#ReturnedList) O

\* Now go through the returned list and add the entries to the visible #Emp\_Lis

Clr\_List Named(#EMP\_LIST)

For Each(#EmployeeObject) In(#ReturnedList)

Change Field(#EMPNO) To('#EMPLOYEEOBJECT.PEMPNO') Change Field(#GIVENAME) To('#EMPLOYEEOBJECT.PGIVENAME') Change Field(#SURNAME) To('#EMPLOYEEOBJECT.PSURNAME') Change Field(#POSTCODE) To('#EMPLOYEEOBJECT.PPOSTCODE') Change Field(#DEPTMENT) To('#EMPLOYEEOBJECT.PDEPARTMENT') Change Field(#SALARY) To('#EMPLOYEEOBJECT.PSALARY')

Add\_Entry To\_List(#EMP\_LIST)

Endfor

Endroutine

End\_Com ↑ Sorted Array Collection Example

## **6.22.5 Collection Programming Summary**

Here is a summary of how you add and access items in a collection. These techniques are described in more detail in the examples for different collection types.

Add Items to Collections

Access Collection Items

1 6. Creating Applications Using Components

# Add Items to Collections

How you add items to collections depends on the type of collection you are using.

To add items to a keyed collection with Style(Factory) you simply refer to an item in the collection. If an item with the specified key does not exist, it is created automatically. For example setting the properties of button #Button\_Factory<1> creates it if it does not already exist:

Set Com(#Button\_Factory<1> Parent(#Com\_Owner) Height(20) Width(100) L

To add items to a keyed collection with Style(Collection) and to any other ordered collection such as array or list collection, you explicitly create the object either with a SET\_REF \*create\_as or with a DEFINE\_COM statement and a SET\_REF.

This code adds a button to a keyed collection with Style(Collection):

```
Set_Ref Com(#Button_Collection<#Use_Key>) To(*create_as #Prim_Phbn)
Set Com(#Button_Collection<#Use_Key>) Parent(#Com_Owner) Height(20) '
```

This method routine creates buttons:

Mthroutine Name(CreateButton) Define\_Map For(\*Input) Class(#Std\_Num) Name(#TheKey) Define\_Com Class(#Prim\_phbn) Name(#Button) Set\_Ref Com(#Button\_Collection<#TheKey>) To(#Button) Endroutine

All other collections except keyed collections have Insert methods to add items to the collection. The item to be inserted in the collection has to be created before it is inserted. This code adds #EmployeeNumbers to a collection named #PayLoadList:

Define\_Com Class(#Prim\_ALPH) Name(#EmployeeNumber) Reference(\*Dyr

Set\_Ref Com(#EmployeeNumber) To(\*Create\_as #Prim\_ALPH) Set Com(#EmployeeNumber) Value(#EmpNo) Invoke Method(#PayLoadList.Insert) Item(#EmployeeNumber)

↑ 6.22.5 Collection Programming Summary

# **Access Collection Items**

FOR/ENDFOR Collection Properties Read-Only Access to Collections Source Code for the Collection Programming Example 1 6.22.5 Collection Programming Summary

# FOR/ENDFOR

You can access the items in collections using a For/EndFor loop.

This example moves all the forms in a ChildForms collection to the left by 5 pixels:

For Each(#Current) In(#ChildForms) Change Field(#NEW\_LEFT) To('#Current.Left - 5') Set Com(#Current) Left(#New\_Left) Endfor

This example disables all the items in submenu #smnu\_1:

```
For Each(#Current) In(#smnu_1.Items)
Set Com(#Current) Enabled(False)
Endfor
```

# **Collection Properties**

Collections have a number of useful properties that can provide information about the collection items.

Sequential collections such as List and Array, provide the following:

First	The first item in the collection. #Com_Owner.Caption := #Collection.First.ExampleProperty
	This item could also be accessed as follows #Com_Owner.Caption := #Collection<1>.ExampleProperty
Last	The last available instance in the collection #Com_Owner.Caption := #Collection.Last.ExampleProperty

**IndexOf** Returns the position of the supplied object in the collection #Std\_num := #Collection.IndexOf<#Object>

Ordered collections, such as Keyed and Dictionary collections, have no real concept of first and last. Instead a "Key" concept is used:

**KeyOf** The value or object used to provide access to the collection item For Each(#Item) In(#KeyedCollection) #Empno := #KeyedCollection.Keyof<#Item>

Endfor

Collections provide a wide variety of methods and properties to allow you to easily manipulate the collection content. You are encouraged to investigate these methods and you will find details in the Visual LANSA Feature Help (F1 help) or Feature Help Guide.

# **Read-Only Access to Collections**

Source Code for the Collection Programming Example

To establish automatic read-only access to a collection you can use the DEFINE\_PTY reference to the collection using the \*Collection keyword for the GET parameter.

This statement creates a collection:

Define\_Com Class(#PRIM\_ACOL<#PRIM\_PHBN>) Name(#COLLECTION)

And this statement creates a read-only reference to it:

Define\_Pty Name(TheCollection) Get(\*Collection #Collection)

You can use these kinds of properties to access a collection to do look-ups or to loop through the collection contents. This property does not allow you to change the contents of the collection. The collection reference can also be used to allow other owner components access to the collection without having to create a payload and pass references.

In this example selected buttons are added to a collection when the Insert button is clicked. The Iterate Collection button iterates the actual collection and adds its contents to the list view, the Iterate Reference button accesses the contents of the collection using the reference set in the property and adds its contents to the list view (so the displayed items are always the same):

One	Add me	Insert	Key	Caption
Two	Add me	Remove All	0000001 0000002	Two Three
Three	Add me	Iterate Collectic		
Four	Add me	Iterate Referenc		
Five	Add me			

### Source Code for the Collection Programming Example

\* \* \* CON

```
* COMPONENT: STD_FORM
```

\*

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(261) Clientwidth

Define Com Class(#PRIM GPBX) Name(#GPBX 1) Caption('Define Pty \*C Define Com Class(#PRIM PHBN) Name(#PHBN 1) Caption('One') Display Define Com Class(#PRIM CKBX) Name(#CKBX 1) Caption('Add me') Dist Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_2) Caption('Two') Display Define\_Com Class(#PRIM\_CKBX) Name(#CKBX\_2) Caption('Add me') Dist Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_3) Caption('Three') Displa Define Com Class(#PRIM CKBX) Name(#CKBX 3) Caption('Add me') Dist Define Com Class(#PRIM PHBN) Name(#PHBN 4) Caption('Four') Display Define Com Class(#PRIM CKBX) Name(#CKBX 4) Caption('Add me') Dist Define Com Class(#PRIM PHBN) Name(#PHBN 5) Caption('Five') Display Define Com Class(#PRIM CKBX) Name(#CKBX 5) Caption('Add me') Dist Define Com Class(#PRIM GPBX) Name(#GPBX 3) Caption('Collection') Di Define Com Class(#PRIM PHBN) Name(#INSERT) Caption('Insert') Display Define Com Class(#PRIM PHBN) Name(#REMOVEALL) Caption('Remove Define\_Com Class(#PRIM\_PHBN) Name(#ITERATECOLL) Caption('Iterate Define Com Class(#PRIM PHBN) Name(#ITERATEREF) Caption('Iterate R

Define\_Com Class(#PRIM\_GRID) Name(#GRID\_1) Captionnoblanklines(Tru Define\_Com Class(#PRIM\_GDCL) Name(#GDCL\_1) Caption('Key') Caption Define\_Com Class(#PRIM\_GDCL) Name(#GDCL\_2) Caption('Caption') Caption Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1)

Define\_Pty Name(Button1) Get(\*Reference #Phbn\_1) Define\_Pty Name(Button2) Get(\*Reference #Phbn\_2) Define\_Pty Name(Button3) Get(\*Reference #Phbn\_3) Define\_Pty Name(Button4) Get(\*Reference #Phbn\_4) Define\_Pty Name(Button5) Get(\*Reference #Phbn\_5) Define\_Com Class(#PRIM\_ACOL<#PRIM\_PHBN>) Name(#COLLECTION)
Define\_Pty Name(TheCollection) Get(\*Collection #Collection)

Evtroutine Handling(#INSERT.Click) If Cond('#ckbx 1.ButtonState = Checked') Invoke Method(#Collection.Insert) Item(#PHBN\_1) Endif If Cond('#ckbx\_2.ButtonState = Checked') Invoke Method(#Collection.Insert) Item(#PHBN\_2) Endif If Cond('#ckbx\_3.ButtonState = Checked') Invoke Method(#Collection.Insert) Item(#PHBN\_3) Endif If Cond('#ckbx\_4.ButtonState = Checked') Invoke Method(#Collection.Insert) Item(#PHBN\_4) Endif If Cond('#ckbx\_5.ButtonState = Checked') Invoke Method(#Collection.Insert) Item(#PHBN 5) Endif Endroutine Evtroutine Handling(#REMOVEALL.Click) Invoke Method(#Collection.RemoveAll) Endroutine Evtroutine Handling(#ITERATECOLL.Click)

Clr\_List Named(#GRID\_1)

For Each(#Current) In(#Collection) Key(#CurrentKey)

Change Field(#STD\_COUNT) To('#CurrentKey.Value') Change Field(#STD\_DESCS) To('#Current.Caption')

Add\_Entry To\_List(#GRID\_1)

Endfor

Endroutine Evtroutine Handling(#ITERATEREF.Click) Clr\_List Named(#GRID\_1)

For Each(#Current) In(#Com\_Owner.TheCollection) Key(#CurrentKey)

Change Field(#STD\_COUNT) To('#CurrentKey.Value') Change Field(#STD\_DESCS) To('#Current.Caption')

Add\_Entry To\_List(#GRID\_1)

Endfor

Endroutine End\_Com

# 6.23 Referencing

You can use references to:

- Dynamically create and destroy components. In this way you can control the life-span of components and the memory use of your application. For information, refer to 6.23.1 Dynamic Multi-Form Application.
- Pass values between components. Referencing is the most efficient way of communicating between components.
- Communicate with ActiveX (COM) components. For details, refer to ActiveX Controls.

References are set and used in:

- The REFERENCE parameter of the DEFINE\_COM command
- The SET\_REF command
- The IF\_REF command
- In the PASS \*BY\_REFERENCE parameter of the DEFINE\_MAP command
- ↑ 6.23 Referencing

# 6.23.1 Dynamic Multi-Form Application

This example differs from the example shown in 6.19 Multi-Form Applications in that a reference is explicitly set to the child form and this reference is destroyed when the child form is closed. This appoach has the benefit of allowing the developer to have control over the life span (and thus the memory usage) of components.

A main form can display a child form showing address information for a selected employee.

mployee Number A	0070	Show Addi	ress Form	
Employee Number		Employee Surname		
A0070	Brown		<u> </u>	
A0090	Blogg	s v		
A0193	Smith	son		
A0907	Simps	ion		
A1001	Jon	Address Form		
A1002	Sm			
A1003	Sm			100
A1004	Sm	Employee Surname	Brown	
A1005	Sm	Employee Number	A0070	
A1006	Sm			
		Street No and Name	12 Railway Street	
		Suburb or Town	Baulkham Hills	
		State and Country	NSW	

The Main Form creates a reference to the Address Form when the Show Address Form button is clicked. When the Address Form is closed, an event is triggered to the Main Form which destroys the reference to the Address Form.

↑ 6.23 Referencing

### **Define the Reference**

The main form defines a reference to the child form (in this example called #REFEX2):

DEFINE\_COM class(#REFEX2) name(#AddressForm) reference(\*dynamic)

When we use REFERENCE(\*DYNAMIC) we tell the application that an instance of the form #REFEX2, referred to as #AddressForm, will be created when we get round to it, not when the main form is loaded (as is the case normally with a DEFINE\_COM statement). This means that before we can use this new reference, we have to create it using a SET\_REF command.

### **Create the Reference**

Show Address Form

A reference to the AddressForm is created explicitly in the click event of the Show Address Form button. The reference creates an instance of the AddressForm:

SET\_REF com(#AddressForm) to(\*create\_as #refex2)

After this statement we are in basically the same position as we would have been had we simply used a normal DEFINE\_COM without the dynamic reference.

Now that an instance of the AddressForm exists, we can run the methods in it. We invoke u\_GetDetails passing in to the AddressForm the employee number anad surname:

INVOKE method(#AddressForm.u\_GetDetails) U\_WITHEMPNO(#EMPNO)

This method in AddressForm receives the employee number and surname, retrieves address information and then displays the form:

MTHROUTINE name(u\_GetDetails) DEFINE\_MAP for(\*INPUT) class(#EMPNO) name(#u\_WithEMPNO) DEFINE\_MAP for(\*INPUT) class(#SURNAME) name(#u\_WithSURNAME) CHANGE field(#EMPNO) to('#U\_WITHEMPNO.VALUE') CHANGE field(#SURNAME) to('#U\_WITHSURNAME.VALUE')

IF cond('#EMPNO \*NE \*BLANKS') FETCH fields(#ADDRESS1 #ADDRESS2 #ADDRESS3) from\_file(PSLMST ENDIF INVOKE method(#COM\_OWNER.ShowForm) ENDROUTINE

### **Destroy the Reference**

When the AddressForm is closed, it signals a user-defined event:

SIGNAL event(u\_detail\_form\_closed)

The MainForm has an event routine which handles this event by setting the reference to the AddressForm to \*NULL:

EVTROUTINE handling(#AddressForm.u\_Detail\_form\_closed) SET\_REF com(#AddressForm) to(\*null) ENDROUTINE

This means that when AddressForm closes, it is removed from memory. The form itself as well as the button and the fields on it all disappear from the system.

# **Check that Reference Has Been Destroyed**

A simple test to verify that references have been destroyed is to put this line of code immediately after setting the reference to AddressForm to \*NULL:

INVOKE method(#AddressForm.u\_GetDetails) U\_WITHEMPNO(#EMPNO)

It will attempt to reference the instance of AddressForm, but will find nothing. The application will then crash as it cannot find what it is looking for:

Fatal Er	ror
8	Component : REFEX1 Main Form Statement : 37 Message : Null or invalid component reference returned by expression evaluation. Routine : File LPCRDISP at line 1764.
	ОК

### Note

The fact that the application crashes when the object it tries to reference does not exist should alert you to the fact that if you go down the route of controlling the life of child and related forms, you must do it completely. LANSA shields the developer from a lot of the complexity of writing OO Windows applications, and many of the difficult procedures required in for example C and C++ are avoided. If you start controlling the life of components, you have to be aware of the implications.

That being said, controlling references is often required in complex and efficient windows applications.

# Source Code for the Dynamic Reference Forms

#### **Main Form**

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #PRIM\_FORM) CAPTION('Main Form') HE \* When we use REFERENCE(\*DYNAMIC) we tell the application that an ins \* #REFEX2 (referred to as #AddressForm) will be created when we get round \* This means that before we can use this new reference, we have to create it us DEFINE\_COM class(#REFEX2) name(#AddressForm) reference(\*dynamic)

DEFINE\_COM class(#PRIM\_PHBN) name(#PHBN\_ADDR) CAPTION('Sho DEFINE\_COM class(#PRIM\_STBR) name(#STBR\_1) DISPLAYPOSITION( DEFINE\_COM class(#EMPNO.Visual) name(#EMPNO) DISPLAYPOSITIOI DEFINE\_COM class(#PRIM\_GRID) name(#GRID\_1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_GDCL) name(#GDCL\_1) DISPLAYPOSITION DEFINE\_COM class(#PRIM\_GDCL) name(#GDCL\_2) DISPLAYPOSITION

EVTROUTINE handling(#com\_owner.Initialize) SELECT fields(#GRID\_1) from\_file(PSLMST) ADD\_ENTRY to\_list(#GRID\_1) ENDSELECT ENDROUTINE

EVTROUTINE handling(#PHBN\_ADDR.Click)

\* Create an instance of the AddressForm. After this statement we will be in ba \* we simply used a normal DEFINE\_COM without the dynamic reference: SET\_REF com(#AddressForm) to(\*create\_as #refex2)

\* Now that an instance of the AddressForm exists, we can run the methods in i \* number and surname:

INVOKE method(#AddressForm.u\_GetDetails) U\_WITHEMPNO(#EMPNO) ENDROUTINE

\* This event signalled from the AddressForm tells this form that it has closed: EVTROUTINE handling(#AddressForm.u\_Detail\_form\_closed)

\* When the instance of the AddressForm closes, remove it from memory by se

\* This says remove anything related to AddressForm from memory. In other w

\* and the fields will all disappear from the system:

SET\_REF com(#AddressForm) to(\*null)

#### ENDROUTINE END\_COM

↑ Source Code for the Dynamic Reference Forms

#### **Address Form**

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #PRIM\_FORM) CAPTION('Address Form') DEFINE\_COM class(#PRIM\_STBR) name(#STBR\_1) DISPLAYPOSITION( DEFINE\_COM class(#EMPNO.Visual) name(#EMPNO) DISPLAYPOSITIOI DEFINE\_COM class(#SURNAME.Visual) name(#SURNAME) DISPLAYPO DEFINE\_COM class(#PRIM\_PHBN) name(#PHBN\_1) CAPTION('Close') D DEFINE\_COM class(#ADDRESS1.Visual) name(#ADDRESS1) DISPLAYPC DEFINE\_COM class(#ADDRESS2.Visual) name(#ADDRESS2) DISPLAYPC DEFINE\_COM class(#ADDRESS3.Visual) name(#ADDRESS3) DISPLAYPC

\* Define an event which will tell this form has closed DEFINE\_EVT name(u\_Detail\_form\_closed)

\* Published method which will get the #empno and #surname values from the \* from the PSLMST file and then show this form MTHROUTINE name(u\_GetDetails) DEFINE\_MAP for(\*INPUT) class(#EMPNO) name(#u\_WithEMPNO) DEFINE\_MAP for(\*INPUT) class(#SURNAME) name(#u\_WithSURNAME) CHANGE field(#EMPNO) to('#U\_WITHEMPNO.VALUE') CHANGE field(#SURNAME) to('#U\_WITHSURNAME.VALUE')

IF cond('#EMPNO \*NE \*BLANKS') FETCH fields(#ADDRESS1 #ADDRESS2 #ADDRESS3) from\_file(PSLMST ENDIF INVOKE method(#COM\_OWNER.ShowForm) ENDROUTINE

EVTROUTINE handling(#PHBN\_1.Click) INVOKE method(#com\_owner.closeform) ENDROUTINE

EVTROUTINE handling(#COM\_OWNER.Closing) options(\*NOCLEARMES \* signal an event to the parent form telling that this form has closed SIGNAL event(u\_detail\_form\_closed) ENDROUTINE END\_COM ↑ Source Code for the Dynamic Reference Forms

# 6.24 Data Classes

Data classes are components that define a value but are not visible. In most cases a data class corresponds to a repository field.

When you drag a field from the repository to a form, you create a visualization of the field. To try this out, drag the field #SALARY to a form. This definition is created for the field:

DEFINE\_COM class(#SALARY.Visual) name(#SALARY)

Note that the class here is #SALARY.Visual. The Visual class creates a component which displays a value and can accept user input. In the Details tab of the editor you can see all the properties of the Visual class. Many of them control the way the field is displayed.

Now, delete the Visual qualifier from the name of the class:

```
DEFINE_COM class(#SALARY) name(#SALARY)
```

This changed statement now defines the data class component of #SALARY. The field is no longer displayed on the form (because it is not visualized).The data class component simply describes a value, in other words the basic characteristics of the component: its data type, length, decimals etc.

Display the Details tab to see the properties of the #SALARY data class. Note that it has very few properties compared to the properties of the #SALARY.Visual class.

Details	
SALARY	<b>*</b>
<b>□</b> • <b>?</b>	
Properties Events Methods	
ComponentClassName	SALARY
ComponentPatternName	SALARY
ComponentTag	
ComponentTypeName	SALARY
Pecimals	2
🖻 Length	11
interest and the second	SALARY
Cwner 🖉	#DEMCOM18
Parent 🦉	#DEMCOM18
🖻 Text	#DEMCOM18
🖉 UsePicklist	True
P Value	.00

# 6.24.1 Using a Data Class

6.24.2 Data Classes as Variables

 $\ensuremath{\Uparrow}$  6. Creating Applications Using Components

### 6.24.1 Using a Data Class

You can use a data class component:

ToWhen you want to create a simple variable which has thecreate acharacteristics (type, properties etc.) of a field or a primitive datavariableclass.

See 6.24.2 Data Classes as Variables.

In a spin<br/>edit boxWhen you want a numeric field to be displayed with spin edit<br/>buttons, add a spin edit box to your form and then specify the field<br/>as the value of the DataClass property of the spin edit.For example, if you add the field #SALARY to a form, it is<br/>displayed as a plain input/output field. If you want this field to be<br/>displayed with spin buttons, you must first add a spin edit box to<br/>your component and then assign the field #SALARY as the<br/>DataClass property of the spin edit. The spin edit box will then<br/>behave exactly like the #SALARY field apart from the spin buttons.Spin buttons are used to increment numbers and therefore only a<br/>numeric field can be used as the value of the DataClass property in<br/>a spin edit box.

In a<br/>comboWhen you want to apply the characteristics (such as type and<br/>length) and rules of a field to the value entered in the edit portion of<br/>the combo box, specify the field as the value of the combo box's<br/>DataClass property.

For example, if you want the value entered in the combo box to be a numeric value accepted by the #SALARY field, make #SALARY the value of the combo box's DataClass property.

Another, less common use for the DataClass property of a combo box is to show the value of a specified field in the edit portion of the combo box.

Normally the edit portion of the combo box shows the value of the current item of the first column (DisplayPosition 1). If you want the value of another field than the one that was used as the source of its first column to be shown in the edit area, specify this field as its

DataClass. This field can be one of the fields used as the source of other columns in the combo box or any other field in the repository. Normally you would not specify any of the fields used as columns in the combo box as the DataClass value because if you want a column displayed in the edit area, it is simpler to change its DisplayPosition to 1. A more typical usage would be to create a field that combines the current values of several columns and use this field as the DataClass so that all the values are visible in the edit area.

For example if the combo box has a column for the first name and surname of employees, a field that concatenates the values of these two fields could be used as the DataClass.

In a When you want to create property sheet entries. A data class needs to be assigned to every entry in the property sheet to handle how the value for the entry is displayed and modified. See 6.26 Property Sheets.

↑ 6.24 Data Classes

# 6.24.2 Data Classes as Variables

You can define data class components based on any field enrolled in the repository. Here are some examples of how you would define your own data class components:

DEFINE\_COM class(#GIVENAME) name(#USER\_GIVENAME) DEFINE\_COM class(#EMPNO) name(#LAST\_EMPNO) DEFINE\_COM class(#SALARY) name(#SALARY) DEFINE\_COM class(#SALARY) name(#OLD\_SALARY) DEFINE\_COM class(#SALARY) name(#NEW\_SALARY)

And here is how you would use commands to manipulate the components:

SET com(#SALARY) value(34000.56) SET com(#OLD\_SALARY) value(12567.45) SET com(#NEW\_SALARY) value(#OLD\_SALARY.Value) IF cond('#SALARY.Value <= #OLD\_SALARY.Value')

In the above #SALARY, #OLD\_SALARY and #NEW\_SALARY are all components of the class #SALARY. The class defines what they are, how long they are, what number of decimals they have, etc.

The important thing to note is that #SALARY in the LANSA repository defines a class of an object. It does not define a component instance. The instances are defined in the DEFINE\_COM statements and there may be many instances with many different names . By default a component has the same name as the class to make coding easier.

This concept is relatively easy to understand when dealing with your own things such as #EMPNO, #SALARY, #GIVENAME because they are such simple things.

However, your repository contains many other classes shipped with LANSA. These components are sometimes referred to as 'primitives' and their names start with #PRIM\_. For example #PRIM\_ALPH is a primitive alphanumeric data class and #PRIM\_NMBR is a primitive numeric data class. The primitive components can be used like your own classes (or fields) to define components within your code.

For example to define a string variable in your code, you could define a

#MYSTRING component based on the primitive alphanumeric variable:

DEFINE\_COM class(#PRIM\_ALPH) name(#MYSTRING)

And then use this component in your code:

SET com(#MYSTRING) value(#EMPNO)

Similarly, to define a numeric variable in your code, you could define a #MYNUMBER component based on the primitive numeric data class:

DEFINE\_COM class(#PRIM\_NMBR) name(#MYNUMBER)

The primitive data classes have also some special uses such as to create 6.26.2 Picklists in 6.26 Property Sheets. 6.25 Using Common Dialogs

↑ 6.24 Data Classes

# **6.25 Using Common Dialogs**

The standard Windows dialogs Open, Save and Color are available in Visual LANSA components:

Color	×
Basic colors:	
<u>C</u> ustom colors:	
Define Custom Colors	>>
OK Cancel	

The common dialogs are defined as interfaces of the Desktop Application component (PRIM\_APPL):

DEFINE\_COM CLASS(#PRIM\_APPL.ICommonDialogColor) NAME(#Colo

-0-	ICommonDialogColor
ĝ	ICommonDialogFileOpen
<b>6</b>	ICommonDialogFileSave
<b>6</b>	ICommonDialogFolderBrowse
<b>6</b>	IDragDropFilePaths
<b>6</b>	IMonitor
<b>6</b>	IMonitors
Cla	asses

They are instantiated and run using the SYS\_APPLN component, for example:

INVOKE METHOD(#sys\_appln.CreateColorDialog) RESULT(#COLORDLG

The dialog features can be obtained using Command Assistant or Auto Complete, or Feature Help.

You can copy and execute these sample forms to see how to use the common dialogs:

- 6.25.1 Color Dialog Sample Form
- 6.25.2 Save Dialog Sample Form
- 6.25.3 Open Dialog Sample Form
- 1 6. Creating Applications Using Components

### 6.25.1 Color Dialog Sample Form

Copy and paste this code to a form to execute a sample application that invokes the Color Dialog. The form must be enabled for full RDLMX.

```
*
*
*
COMPONENT: STD_FORM
*
*
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) CLIENTHEIGHT(459) CLIE]
Define_Com Class(#PRIM_LABL) Name(#InitialValuesLbl) CAPTION('Colo
Define_Com Class(#PRIM_LABL) Name(#InitialColorLbl) CAPTION('Initial
Define_Com Class(#PRIM_EDIT) Name(#InitialColor) DISPLAYPOSITION(
```

Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) CAPTION('Start Color

Define\_Com Class(#PRIM\_APPL.ICommonDialogColor) Name(#colorDlg) F

Define\_Com Class(#PRIM\_LABL) Name(#ResultLbl) CAPTION('Color dialc Define\_Com Class(#PRIM\_LABL) Name(#ResultColorLbl) CAPTION('ResultColorLbl) CAP

Define\_Com Class(#PRIM\_PHBN) Name(#GotoOpenFileForm) CAPTION('C Define\_Com Class(#PRIM\_EDIT) Name(#ResultColor) DISPLAYPOSITION Define\_Com Class(#PRIM\_GPBX) Name(#GPBX\_1) DISPLAYPOSITION(9 Define\_Com Class(#PRIM\_RDBN) Name(#OkButtonPressed) CAPTION('Ok Define\_Com Class(#PRIM\_RDBN) Name(#CancelButtonPressed) CAPTION( Define\_Com Class(#PRIM\_CKBX) Name(#FullDetailsMode) CAPTION('FullDefine\_Com Class(#PRIM\_CKBX) Name(#FullDetailsMode) CAPTION('FullDetailsMode) CAPTION('FullDetailsMode)

Define\_Com Class(#AAOPENFIL) Name(#OpenFileDlgForm) Evtroutine Handling(#com\_owner.Initialize) Set Com(#com\_owner) CAPTION(\*component\_desc) Endroutine

Evtroutine Handling(#PHBN\_1.Click) Invoke Method(#sys\_appln.CreateColorDialog) RESULT(#COLORDLG)

Define Field(#boolRes) Type(\*BOOLEAN)

#COLORDLG.ChosenColor := green
If (#FULLDETAILSMODE.ButtonState = checked)
#COLORDLG.FullyOpen := true
Else
#COLORDLG.FullyOpen := false
Endif

Invoke Method(#COLORDLG.Show) OKPRESSED(#boolRes) FORMOWNE

If (#BOOLRES) #OKBUTTONPRESSED.ButtonChecked := true #CANCELBUTTONPRESSED.ButtonChecked := false

#RESULTCOLOR.Value := #COLORDLG.ChosenColor

Else #CANCELBUTTONPRESSED.ButtonChecked := true #okbUTTONPRESSED.ButtonChecked := false

#RESULTCOLOR.Value := "

Endif Endroutine Evtroutine Handling(#GotoOpenFileForm.Click) Invoke Method(#OPENFILEDLGFORM.ShowForm) Endroutine End\_Com

1 6.25 Using Common Dialogs

### 6.25.2 Save Dialog Sample Form

Copy and paste this code to a form to execute a sample application that invokes the Save Dialog. The form must be enabled for full RDLMX.

```
*
*
COMPONENT: STD_FORM
*
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) CLIENTHEIGHT(686) CLIE]
Define_Com Class(#PRIM_LABL) Name(#InitialValuesLbl) CAPTION('Save
Define_Com Class(#PRIM_LABL) Name(#InitialSaveFileLbl) CAPTION('Ini
```

Define\_Com Class(#PRIM\_EDIT) Name(#InitialDirectory) DISPLAYPOSITI

Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) CAPTION('Start Save I

Define\_Com Class(#PRIM\_APPL.ICommonDialogFileSave) Name(#saveFile)

Define\_Com Class(#PRIM\_LABL) Name(#ResultLbl) CAPTION('Save File d

Define\_Com Class(#PRIM\_GPBX) Name(#GPBX\_1) DISPLAYPOSITION(7 Define\_Com Class(#PRIM\_RDBN) Name(#OkButtonPressed) CAPTION('Ok Define\_Com Class(#PRIM\_RDBN) Name(#CancelButtonPressed) CAPTION( \* Explorer mode has no effect on Save File - it is a Windows problem so it has \* DEFINE\_COM CLASS(#PRIM\_CKBX) NAME(#ExplorerMode) CAPTIO

Define\_Com Class(#AAsavefil) Name(#SaveFileDlgForm) Define\_Com Class(#PRIM\_EDIT) Name(#Title) DISPLAYPOSITION(8) LEF Define\_Com Class(#PRIM\_LABL) Name(#TitleLbl) CAPTION('Title') DISPI Define\_Com Class(#PRIM\_CKBX) Name(#overwritePrompt) CAPTION('Ove

Define\_Com Class(#PRIM\_LTBX) Name(#fileLst) DISPLAYPOSITION(2) H Define\_Com Class(#PRIM\_LBCL) Name(#TRACECOL1) DISPLAYPOSITI( Define\_Com Class(#PRIM\_EDIT) Name(#File) DISPLAYPOSITION(11) LEI Define\_Com Class(#PRIM\_LABL) Name(#FileLb) CAPTION('File') DISPLA

Evtroutine Handling(#com\_owner.Initialize)

Set Com(#com\_owner) CAPTION(\*component\_desc) Endroutine

Evtroutine Handling(#PHBN\_1.Click) Invoke Method(#sys\_appln.CreateFileSaveDialog) RESULT(#SaveFileDLG)

Define Field(#boolRes) Type(\*BOOLEAN)

Clr\_List Named(#filelst)

#SaveFileDLG.Title := #TITLE.Text
#saveFileDLG.InitialDir := #INITIALDIRECTORY.Text
#saveFILEDLG.File := #FILE.Text

#saveFileDLG.AddFilter( 'First entry (\*.nothing)' '\*.nothing' )

\* deliberately make .cpp as .hpp to check overwriting facility later #saveFileDLG.AddFilter( 'C++ defn files (\*.cpp)' '\*.hpp' ) #saveFileDLG.AddFilter( 'C++ header files ( ended with \*.hpp)' '\*.hpp' )

```
* correct the first the initial filter
#saveFileDLG.FilterCaption<2> := 'C++ defn files (corrected to *.cpp & *.c)'
#saveFileDLG.FilterExtension<2> := '*.cpp;*.c'
* add an extra filter
#saveFileDLG.FilterCaption<4> := 'Text Files (*.txt)'
#saveFileDLG.FilterExtension<4> := '*.txt'
* select something non-default
#saveFileDLG.FilterIndex := 2
```

If (#overwriteprompt.ButtonState = checked) #saveFILEDLG.OverwritePrompt := true Else #saveFILEDLG.OverwritePrompt := false Endif

\* Explorer mode has no effect on Save File - it is a Windows problem so it has

```
* IF (#Explorermode.ButtonState = checked)
```

```
* #saveFILEDLG.ExplorerStyle := true
```

```
* ELSE
```

\* #saveFILEDLG.ExplorerStyle := false \* ENDIF

Invoke Method(#saveFileDLG.Show) OKPRESSED(#boolRes) FORMOWNE

If (#BOOLRES) #OKBUTTONPRESSED.ButtonChecked := true #CANCELBUTTONPRESSED.ButtonChecked := false

```
Define Field(#i) Type(*INT)
#i := 1
#std_text := #saveFILEDLG.FileTitle
Add_Entry To_List(#fileIst)
Else
#CANCELBUTTONPRESSED.ButtonChecked := true
#okbUTTONPRESSED.ButtonChecked := false
```

Endif Endroutine End\_Com

↑ 6.25 Using Common Dialogs

### 6.25.3 Open Dialog Sample Form

Copy and paste this code to a form to execute a sample application that invokes the Open Dialog. The form must be enabled for full RDLMX.

```
*
*
COMPONENT: STD_FORM
*
*
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) CLIENTHEIGHT(686) CLIE]
Define_Com Class(#PRIM_LABL) Name(#InitialValuesLbl) CAPTION('Oper
Define_Com Class(#PRIM_LABL) Name(#InitialOpenFileLbl) CAPTION('In
```

Define\_Com Class(#PRIM\_EDIT) Name(#InitialDirectory) DISPLAYPOSITI

Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) CAPTION('Start Open

Define\_Com Class(#PRIM\_APPL.ICommonDialogFileOpen) Name(#openFile

Define\_Com Class(#PRIM\_LABL) Name(#ResultLbl) CAPTION('Open File ( Define\_Com Class(#PRIM\_LABL) Name(#FileCountLb) CAPTION('Selected

Define\_Com Class(#PRIM\_PHBN) Name(#GotoOpenFileForm) CAPTION('C Define\_Com Class(#PRIM\_EDIT) Name(#FileCount) DISPLAYPOSITION(9 Define\_Com Class(#PRIM\_GPBX) Name(#GPBX\_1) DISPLAYPOSITION(1 Define\_Com Class(#PRIM\_RDBN) Name(#OkButtonPressed) CAPTION('Ok Define\_Com Class(#PRIM\_RDBN) Name(#CancelButtonPressed) CAPTION( Define\_Com Class(#PRIM\_CKBX) Name(#ExplorerMode) CAPTION('Explo

Define\_Com Class(#AAsavefil) Name(#SaveFileDlgForm) Define\_Com Class(#PRIM\_CKBX) Name(#Multiselect) CAPTION('Multisele Define\_Com Class(#PRIM\_EDIT) Name(#Title) DISPLAYPOSITION(13) LE Define\_Com Class(#PRIM\_LABL) Name(#TitleLbl) CAPTION('Title') DISPI Define\_Com Class(#PRIM\_CKBX) Name(#HideReadOnly) CAPTION('Hide Only') DISPLAYPOSITION(15) LEFT(24) PARENT(#COM\_OWNER) TABI

Define\_Com Class(#PRIM\_LTBX) Name(#fileLst) DISPLAYPOSITION(2) H Define\_Com Class(#PRIM\_LBCL) Name(#TRACECOL1) DISPLAYPOSITI( Define\_Com Class(#PRIM\_EDIT) Name(#File) DISPLAYPOSITION(16) LE Define\_Com Class(#PRIM\_LABL) Name(#FileLb) CAPTION('File') DISPLA

Evtroutine Handling(#com\_owner.Initialize) Set Com(#com\_owner) CAPTION(\*component\_desc) Endroutine

Evtroutine Handling(#PHBN\_1.Click) Invoke Method(#sys\_appln.CreateFileOpenDialog) RESULT(#OpenFileDLG)

Define Field(#boolRes) Type(\*BOOLEAN)

#OpenFileDLG.Title := #TITLE.Text #OpenFileDLG.InitialDir := #INITIALDIRECTORY.Text #OPENFILEDLG.File := #FILE.Text

#OpenFileDLG.AddFilter( 'First entry (\*.nothing)' '\*.nothing' )

\* deliberately make .cpp as .hpp to check overwriting facility later #OpenFileDLG.AddFilter( 'C++ defn files (\*.cpp)' '\*.hpp' ) #OpenFileDLG.AddFilter( 'C++ header files ( ended with \*.hpp)' '\*.hpp' )

\* correct the first the initial filter #OpenFileDLG.FilterCaption<2> := 'C++ defn files (corrected to \*.cpp & \*.c)' #OpenFileDLG.FilterExtension<2> := '\*.cpp;\*.c' \* add an extra filter #OpenFileDLG.FilterCaption<4> := 'Text Files (\*.txt)' #OpenFileDLG.FilterExtension<4> := '\*.txt' \* select something non-default #OpenFileDLG.FilterIndex := 2

If (#Hidereadonly.ButtonState = checked) #OPENFILEDLG.HideReadOnly := true Else #OPENFILEDLG.HideReadOnly := false Endif If (#Explorermode.ButtonState = checked) #OPENFILEDLG.ExplorerStyle := true Else #OPENFILEDLG.ExplorerStyle := false Endif If (#Multiselect.ButtonState = checked) #OPENFILEDLG.MultiSelect := true Else #OPENFILEDLG.MultiSelect := false Endif

Invoke Method(#OpenFileDLG.Show) OKPRESSED(#boolRes) FORMOWN

\* ensure fresh start every time the dialog is open Clr\_List Named(#filelst)

If (#BOOLRES) #OKBUTTONPRESSED.ButtonChecked := true #CANCELBUTTONPRESSED.ButtonChecked := false

```
Define Field(#i) Type(*INT)
#i := 1
#std_count := #oPENFileDLG.FileCount
If Cond(#std_count = 0)
#std_text := #OPENFILEDLG.FileTitle
Add_Entry To_List(#fileIst)
Else
Dountil Cond(#i > #std_count)
#std_text := #OPENFileDLG.Files<#i>
Add_Entry To_List(#fileIst)
#i := #i + 1
Enduntil
```

Endif

#FILECOUNT.Value := #OPENFILEDLG.FileCount.AsString

Else #CANCELBUTTONPRESSED.ButtonChecked := true #okbUTTONPRESSED.ButtonChecked := false

Endif

Endroutine Evtroutine Handling(#GotoOpenFileForm.Click) Invoke Method(#SAVEFILEDLGFORM.ShowForm) Endroutine End\_Com

↑ 6.25 Using Common Dialogs

# **6.26 Property Sheets**

The property sheet component is useful for displaying the attributes of an object. A property sheet is a compact way of displaying the attributes for viewing and editing. For example this property sheet displays details for an employee:

First Name	Veronica	
Surname Salary	Brown 60,000.00	
Alpha Options	C	
Numeric Options	34	

Every row (or entry) in the sheet shows the name of a property and its value. The value of an entry in the sheet can be a simple value or it can be a list of values from which the user can choose from. This list is called a picklist:

Salary 60,000.00 Alpha Options C Numeric Options 34 1001	First Name Surname	Veronica Brown
Alpha Options C Numeric Options 34 1001	Salary	
1001	Alpha Options	
	Numeric Options	34
		1001 220
34		

6.26.1 Basics

6.26.2 Picklists

6.26.3 Property Sheet Example

6.27 Keyboard and Mouse Components

# 

### **6.26.1 Basics**

As with any list-type component, start creating your property sheet by dragging the property sheet component to your form or reusable part.

Contro	ols			
(	Common	User Designed	All Controls	
_	Container for othe	er components		*
0	Popup Popup			
	<b>Pop-up Menu</b> Pop-up Menu			
	Progress Bar Progress Bar show	vs how an operation p	progresses	
4	Property Sheet Displays propertie	s of an object		5
	Push Button Performs an actio	n		
۲	Radio Button Radio Button show	ws mutually exclusive	e choices	
	<b>Ribbon</b> Ribbon			
Las .	Shortcut Key Shortcut Key			
J	Sound Plays a sound file			
	<b>Spin Edit Box</b> Spin Edit Box is us	ed to adjust a value		
<u>9</u>	Static Lines and shapes			
-	Status Bar			-

Columns Rows (Entries) 1 6.26 Property Sheets

# Columns

For this example create two fields in the repository to be used as the basis of the columns in the property sheet: #PROPERTY and #VALUE. Make the reference field for both of them #STD\_TEXT.

In the editor, drag these fields from the repository to the property sheet to create a column for the property descriptions and another column for the property values. (As a rule a property sheet should always have two visible columns. In some cases you may want to add other columns which are not visible.)

Note that in a property sheet you do not want the columns to be based on fields with data as you usually do with lists. This is because every row of the property sheet will display a different attribute based on a different field. When you add columns all you are doing is defining the basic structure of the property sheet.

Make the Width of the first column 50 (WidthType Scaleable) and the WidthType of the second column Remainder.

The first column which contains the description of the properties should always be based on an alphanumeric field. The second column should usually also be based on an alphanumeric field (to show the values as strings) except in those rare cases where all entries in the property sheet take numeric values.

↑ 6.26.1 Basics

# **Rows (Entries)**

Salary 60,000.00 Alpha Options C Numeric Options 34	First Name Surname	Veronica Brown	
Alpha Options C			
		34	

To define the First Name entry in the property sheet:

1. In the Initialize event of your property sheet specify a description and a value for the entry (remember the first column is based on the field #PROPERTY and the second column on the field #VALUE):

CHANGE field(#PROPERTY) to("'First Name''') CHANGE field(#VALUE) to(#GIVENAME)

2. Add the entry to the property sheet:

ADD\_ENTRY to\_list(#PROP\_1)

3. Define a Data Class for the entry (in the beginning of your component):

DEFINE\_COM class(#GIVENAME) name(#GIVENAME)

4. In the Initialize event routine assign the data class to the entry. The data class will handle the way the value is displayed and entered:

SET com(#prop\_1.CurrentItem) DATACLASS(#givename)

This statement tells the property sheet that the value for this entry in the second

column in the property sheet should be shown (and edited) according to the class #GiveName. For example, it cannot be more than 20 characters long and it can contain upper or lowercase characters.

You repeat these four basic steps for every entry in the property sheet. For example this is how you would define the Salary entry in the property sheet:

Surname Salary Alpha Options	Brown 60000.00
	60000.00
Apha Options	
	C
Numeric Options	34

1. In the Initialize event of your property sheet specify a description and a value for the entry:

CHANGE field(#PROPERTY) to("'Salary'") CHANGE field(#VALUE) to('#SALARY.TEXT')

Note that because #SALARY is a numeric field but you are displaying it in a column based on an alphanumeric field (#VALUE), you need to display its TEXT property.

2. Add the entry to the property sheet:

ADD\_ENTRY to\_list(#PROP\_1)

3. Define a Data Class for the entry:

DEFINE\_COM class(#SALARY) name(#SALARY)

4. In the Initiallize event routine assign the data class to the entry:

SET com(#prop\_1.CurrentItem) DATACLASS(#salary)

This statement tells the property sheet that the value for this entry in the second column in the property sheet should be shown (and edited) according to the class #SALARY. For example, it's maximum length is 11 digits and it can contain 2 decimals and its default value is 0.

↑ 6.26.1 Basics

### 6.26.2 Picklists

There are two primitive Data Class components in Visual LANSA that you can use in property sheets: PRIM\_ALPH and PRIM\_NMBR. These classes are generic definitions for strings and numbers. In addition, they contain a Picklist property which you can use to display a drop-down of values for a property sheet item, called a picklist.

This is what a picklist looks like:

Salary 60,000.00 Alpha Options B 👱	First Name	Veronica	
Alpha Options <u>B</u>	Surname	Brown	
		60,000.00	
Numeric Options A B C	Alpha Options		ŧ
B C	Numeric Options	A	
		B	
			_

To see what these primitive data classes contain base a data class called #OPTIONSA on #PRIM\_ALPH:

DEFINE\_COM class(#PRIM\_ALPH) name(#OPTIONSA)

Now display the Details tab for the #OPTIONSA data class:

Outline Component Re	epository Command
OPTIONSA - Alphanum	eric data class is a string 🛛 🛨
Properties Events N	1ethods
Case	MixedCase
Length	0
Name	OPTIONSA
Owner	#TEST2
Parent	#TEST2
Picklist	*NULL
Text	
Value	

Notice that the data class component contains a Picklist property. This property

is used to associate a picklist with a property sheet entry. Set Up Picklists Create an Entry with Picklist Retrieve the Selected Picklist Item Picklist Code You Can Copy ↑ 6.26 Property Sheets

# Set Up Picklists

Before you add the entry with picklist, you need to create a collection of picklists, a collection of picklist items and two subroutines, one to add the picklists and one to add the items to the collections. These collections and subroutines handle all the picklists in the property sheet:

1. Create a collection of picklists (#PRIM\_PKLT components):

DEFINE\_COM class(#PRIM\_KCOL) name(#PL\_COL) collects(#PRIM\_PKL

2. Create a collection of picklist items (#PRIM\_PKIT components):

DEFINE\_COM class(#PRIM\_KCOL) name(#PI\_COL) collects(#PRIM\_PKIT

3. Assign index numbers to identify the pick lists and the pick list items:

DEFINE\_COM class(#STD\_COUNT) name(#CUR\_PLIST) DEFINE\_COM class(#STD\_COUNT) name(#CUR\_PITEM)

4. Create a subroutine which creates new picklists by assigning a new index number to the collection:

SUBROUTINE name(ADD\_PLIST) CHANGE field(#CUR\_PLIST) to('#Cur\_Plist + 1') ENDROUTINE

5. Create a subroutine which creates new picklist items by assigning them index numbers and associates the items with the current picklist:

SUBROUTINE name(ADD\_PITEM) parms((#PROPERTY \*RECEIVED) (#\ CHANGE field(#CUR\_PITEM) to('#Cur\_PItem + 1') \* Set the caption and the value of the item plus assign it into the Picklist SET com(#PI\_Col<#Cur\_PItem>) caption(#property) default(True) value(#val ENDROUTINE You can copy the code which sets up picklists from Picklist Code You Can Copy.

# **Create an Entry with Picklist**

Now you have set up the structure for any picklists in the property sheet. Next create an entry for the #OPTIONSA data class (Alpha Options) with an associated picklist:

1. Base the data class for the entry on the elemental data class #PRIM\_ALPH:

DEFINE\_COM class(#PRIM\_ALPH) name(#OPTIONSA)

2. In the Initialize event of your property sheet add the picklist and create the items for it. Then create the entry, associate the pick list with it and add the entry to the property sheet:

```
EXECUTE subroutine(ADD_PLIST)
EXECUTE subroutine(ADD_PITEM) with_parms('A' 'option A')
EXECUTE subroutine(ADD_PITEM) with_parms('B' 'option B')
EXECUTE subroutine(ADD_PITEM) with_parms('C' 'option C')
```

```
CHANGE field(#PROPERTY) to("'Alpha Options''')

* associate the picklist with the #OPTIONSA data class

SET com(#optionsa) picklist(#PL_COL<#Cur_Plist>)

* assign the picklist to the second column

CHANGE field(#VALUE) to('#OPTIONSA.TEXT')

* add the entry

ADD_ENTRY to_list(#PROP_1)

* assign the data class to the entry in the property sheet

SET com(#prop 1.CurrentItem) dataclass(#optionsa)
```

The definition of the Alpha Options entry with its associated picklist is now complete.

### **Retrieve the Selected Picklist Item**

To find out which picklist value was chosen, simply retrieve the the Text or Value property of the data class component on which the entry is based. For example this statement assigns the currently selected value of the Alpha Options entry to #STD\_TEXT:

CHANGE field(#STD\_TEXT) to('#OPTIONSA.TEXT')

# Picklist Code You Can Copy

You can copy and paste the following code to any component in which you want to use property sheets containing picklists. The code creates the collection of picklists and picklist items as well as the subroutines for adding picklists and items to the collections.

Put the following definitions in the beginning of your component:

\* Pick list and pick list item definitions for the property sheet
\* pick list counter (index)
DEFINE\_COM class(#STD\_COUNT) name(#CUR\_PLIST)
\* pick list item counter (index)
DEFINE\_COM class(#STD\_COUNT) name(#CUR\_PITEM)

\* collection of pick lists (shown inside the property sheet)
DEFINE\_COM class(#PRIM\_KCOL) name(#PL\_COL) collects(#PRIM\_PKL
\* collection of pick list items
DEFINE\_COM class(#PRIM\_KCOL) name(#PI\_COL) collects(#PRIM\_PKIT

Put these two subroutines after all the DEFINE\_COM statements in your component:

\* Add new picklist SUBROUTINE name(ADD\_PLIST) \* give the new pick list an index number CHANGE field(#CUR\_PLIST) to('#Cur\_PList + 1') ENDROUTINE

\* Add new picklist item SUBROUTINE name(ADD\_PITEM) parms((#PROPERTY \*RECEIVED) (#\ \* give the new pick list item an index number CHANGE field(#CUR\_PITEM) to('#Cur\_PItem + 1') \* Set the caption and the value of the item plus assign it into the picklist SET com(#PI\_Col<#Cur\_PItem>) caption(#property) default(True) value(#val ENDROUTINE

# 6.26.3 Property Sheet Example

In this example we create a property sheet that shows how you define the list and add entries with both simple values and with picklists:

	Veronica
Surname	Brown
Salary	60,000.00
Alpha Options	C
Numeric Options	34
Surnai Salary Alpha	ame: Veronica me: Brown : 60000.00 Options: C ric Options: 34

Code for the Property Sheet Example

**Property Sheet Notes** 

1 6. Creating Applications Using Components

# **Code for the Property Sheet Example**

Copy and paste this code to a form and then compile and execute the form:

FUNCTION options(\*DIRECT) BEGIN\_COM height(254) left(375) top(161) width(272) DEFINE\_COM class(#PRIM\_PROP) name(#PROP\_1) busyupdatesofparent(T

DEFINE\_COM class(#PRIM\_PHBN) name(#PHBN\_1) caption('Show Proper DEFINE\_COM class(#PRIM\_PRCL) name(#PRCL\_1) displayposition(1) pare DEFINE\_COM class(#PRIM\_PRCL) name(#PRCL\_2) displayposition(2) pare

\* Data class definitions for the property sheet

\* Define #givename data class based on the repository field #givename

DEFINE\_COM class(#GIVENAME) name(#GIVENAME)

\* Define #surname data class based on the elemental data class #PRIM\_ALPH DEFINE\_COM class(#SURNAME) name(#SURNAME)

\* Define #salary data class based on the repository field #salary

DEFINE\_COM class(#SALARY) name(#SALARY)

\* Define #expenses data class based on the elemental data class #PRIM\_NMBE DEFINE\_COM class(#PRIM\_ALPH) name(#OPTIONSA)

\* Define #optionsn data class based on the elemental data class #PRIM\_NMBI DEFINE\_COM class(#PRIM\_NMBR) name(#OPTIONSN) length(9)

\* Pick list and pick list item definitions for the property sheet

\* pick list counter (index)

DEFINE\_COM class(#STD\_COUNT) name(#CUR\_PLIST)

\* pick list item counter (index)

DEFINE\_COM class(#STD\_COUNT) name(#CUR\_PITEM)

\* collection of pick lists (shown inside the property sheet)
DEFINE\_COM class(#PRIM\_KCOL) name(#PL\_COL) collects(#PRIM\_PKL
\* collection of pick list items
DEFINE\_COM class(#PRIM\_KCOL) name(#PI\_COL) collects(#PRIM\_PKIT

\* Add new picklistSUBROUTINE name(ADD\_PLIST)\* give the new pick list an index number

CHANGE field(#CUR\_PLIST) to('#Cur\_PList + 1') ENDROUTINE

\* Add new picklist item SUBROUTINE name(ADD\_PITEM) parms((#PROPERTY \*RECEIVED) (#\ \* give the new pick list item an index number CHANGE field(#CUR\_PITEM) to('#Cur\_PItem + 1') \* Set the caption and the value of the item plus assign it into the picklist SET com(#PI\_Col<#Cur\_PItem>) caption(#property) default(True) value(#val ENDROUTINE

\* fill the property sheet EVTROUTINE handling(#prop\_1.Initialize) options(\*NOCLEARMESSAGES \* set values for the entries CHANGE field(#GIVENAME) to("'Veronica''') SET com(#surname) value('Brown') CHANGE field(#SALARY) to(60000)

\* Add an alphanumeric entry based on the data class of the field - First Name CHANGE field(#PROPERTY) to("'First Name''') CHANGE field(#VALUE) to(#GIVENAME) ADD\_ENTRY to\_list(#PROP\_1) SET com(#prop\_1.CurrentItem) dataclass(#givename)

\* Add an alphanumeric entry based on the data class of the field - Surname CHANGE field(#PROPERTY) to("'Surname''') CHANGE field(#VALUE) to(#surname) ADD\_ENTRY to\_list(#PROP\_1) SET com(#prop\_1.CurrentItem) dataclass(#surname)

\* Add a numeric entry based on the data class of the field - Salary CHANGE field(#PROPERTY) to("'Salary''') CHANGE field(#VALUE) to('#SALARY.TEXT') ADD\_ENTRY to\_list(#PROP\_1) SET com(#prop\_1.CurrentItem) dataclass(#salary)

\* Create picklist with three items and add it to an alphanumeric entry - Alpha ( EXECUTE subroutine(ADD\_PLIST) EXECUTE subroutine(ADD\_PITEM) with\_parms('A' 'option A') EXECUTE subroutine(ADD\_PITEM) with\_parms('B' 'option B') EXECUTE subroutine(ADD\_PITEM) with\_parms('C' 'option C')

CHANGE field(#PROPERTY) to("'Alpha Options''') SET com(#optionsa) picklist(#PL\_COL<#Cur\_PList>) CHANGE field(#VALUE) to('#OPTIONSA.TEXT') ADD\_ENTRY to\_list(#PROP\_1) SET com(#prop\_1.CurrentItem) dataclass(#optionsa)

\* Create picklist with three items and add it to a numeric entry - Numeric Optic EXECUTE subroutine(ADD\_PLIST) EXECUTE subroutine(ADD\_PITEM) with\_parms('1001' '1001') EXECUTE subroutine(ADD\_PITEM) with\_parms('220' '220') EXECUTE subroutine(ADD\_PITEM) with\_parms('34' '34')

CHANGE field(#PROPERTY) to("'Numeric Options''') SET com(#optionsn) picklist(#PL\_COL<#Cur\_PList>) CHANGE field(#VALUE) to('#OPTIONSN.TEXT') ADD\_ENTRY to\_list(#PROP\_1) SET com(#prop\_1.CurrentItem) dataclass(#optionsn)

ENDROUTINE

EVTROUTINE handling(#PHBN\_1.Click) USE builtin(MESSAGE\_BOX\_ADD) with\_args('First name: '#GIVENAME) USE builtin(MESSAGE\_BOX\_ADD) with\_args('Surname: '#SURNAME) USE builtin(MESSAGE\_BOX\_ADD) with\_args('Salary: '#SALARY) USE builtin(MESSAGE\_BOX\_ADD) with\_args('Alpha Options: '#OPTIONS USE builtin(MESSAGE\_BOX\_ADD) with\_args('Numeric Options: '#OPTIO

USE builtin(MESSAGE\_BOX\_SHOW) with\_args(' ' ' ' ' 'Returned Values')

ENDROUTINE END\_COM

1 6.26.3 Property Sheet Example

### **Property Sheet Notes**

### **Assigning the Value**

Note that when you are setting the value of the #Value column and the the data class of the entry is based on #PRIM\_ALPH, #PRIM\_NMBR or a numeric field, you must use the Text property of the component.

CHANGE field(#PROPERTY) to("'Salary''') CHANGE field(#VALUE) to('#SALARY.TEXT') ADD\_ENTRY to\_list(#PROP\_1) SET com(#prop\_1.CurrentItem) dataclass(#salary)

# **Returning a Value**

When you are returning a value from the property sheet, you must use the Value or Text properties of the data classes based on #PRIM\_ALPH or #PRIM\_NMBR:

USE builtin(MESSAGE\_BOX\_ADD) with\_args('First name: '#GIVENAME) USE builtin(MESSAGE\_BOX\_ADD) with\_args('Last name: '#SURNAME) USE builtin(MESSAGE\_BOX\_ADD) with\_args('Salary: '#SALARY) USE builtin(MESSAGE\_BOX\_ADD) with\_args('Alpha Options: '#OPTIONS USE builtin(MESSAGE\_BOX\_ADD) with\_args('Numeric Options: '#OPTIO

1 6.26.3 Property Sheet Example

### 6.27 Keyboard and Mouse Components

You can find out the position of the mouse using the value of the #SYS\_MOUSE component and you can find out whether the Shift, Alt or Ctrl key is pressed down using the #SYS\_KEYBD component.

These two components are not visible in the Repository tab. Typically you would use the values of their properties in a Change command to assign a value to a field:

change #Std\_Num #SYS\_MOUSE.HorPosition

**#SYS\_MOUSE** has these properties:

HorPosition Horizontal position is the number of pixels from the left

VerPosition Vertical position is the number of pixels from the top

**#SYS\_KEYBD** has these properties:

AltKeyDown	True or False. Indicates whether the Alt key is pressed down.
ShiftKeyDown	True or False. Indicates whether the Shift key is pressed down.
ControlKeyDown	True or False. Indicates whether the Control key is pressed down.

1 6. Creating Applications Using Components

# 6.28 Drag-and-Drop

#### Also see

```
6.28.1 6.28.2 Example 1: MovePayload Employees in a Tree
```

6.28.3 Example 2: Move Employees between Two Forms

Drag-and-drop operations are used to move or copy objects from one place to another.

A drag-and-drop operation starts when you select an object with the mouse by clicking the left mouse button. Then you drag the object with the mouse holding the left mouse button down, and finally drop the object by releasing the left mouse button.

A drag-and-drop operation consists of four events:

- StartDrag
- DragOver
- DragDrop
- EndDrag

The object being dragged and dropped is the 6.28.1 Payload.

To enable LANSA controls to accept file paths from Windows Explorer during drag over and drag drop events, enable the following property on Sys\_Appln in (for example) #com\_owner.Initialize.

#SYS\_APPLN.AllowWindowsDragDrop := True

When dragging files, the Payload parameter of the event handlers contains a component interface of type #PRIM\_APPL.IDragDropFilePaths.

Details		C
ETDRAW		
Properties Events Methods		
Auco-Arrange	False	-
🎤 Busyupīdates	Wait	
Parent (Parent	False	
Checko	False	
Columnfight It	20	III
🔎 Columnfáth	End	
Column+entClassNas	True	
ColumnHentDescrip ``	True	
P ColumnResize	True	_
ComponentClassName	PRIM_LTVW	
ComponentPatternName	PRIM_LTVW	
ComponentTag		
ComponentTypeName	PRIM_LTVW	
ComponentVersion	2	
PisplayPosition	1	
PragColumns	False	
DragDropFilePath	True	
	Automatic	
Enabled	True	

The interface exposes two methods:

FileCount to ascertain the number of files being dragged

FilePath<index> to obtain the file path at position index

Following is an example of how this functionality can be used to allow a ListView control to obtain a list of file paths being drag-dropped on it. Create a new form and replace its source with the following code.

Define\_Com Class(#PRIM\_LTVW) Name(#ListView) Columnbuttonheight(2( Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_1) Caption('Items') Caption Define\_Com Class(#PRIM\_LVCL) Name(#LVCL\_2) Caption('File Path') Capt

\* Initialisation Evtroutine Handling(#com\_owner.Initialize) #SYS\_APPLN.AllowWindowsDragDrop := True Set Com(#com\_owner) Caption(\*component\_desc) Endroutine

Evtroutine Handling(#ListView.DragOver) Options(\*NOCLEARMESSAGES If (#dragState \*NE Exit) If (#draggedData \*Is #PRIM\_APPL.IDragDropFilePaths) #STD\_NUM := (#draggedData \*As #PRIM\_APPL.IDragDropFilePaths).FileC #VF\_INSDTA := "

Clr\_List Named(#ListView)

Add\_Entry To\_List(#ListView)

#acceptDrop := True Endif Endif Endroutine

Evtroutine Handling(#ListView.DragDrop) Options(\*NOCLEARMESSAGES Clr\_List Named(#ListView)

If (#draggedData \*Is #PRIM\_APPL.IDragDropFilePaths) Define Field(#NumFiles) Reffld(#STD\_NUM) #NumFiles := (#draggedData \*As #PRIM\_APPL.IDragDropFilePaths).FileCou #STD\_NUM := 1

Dowhile Cond(#STD\_NUM <= #NumFiles) #VF\_INSDTA := (#draggedData \*As #PRIM\_APPL.IDragDropFilePaths).File Add\_Entry To\_List(#ListView) #STD\_NUM += 1 Endwhile Endif Endroutine End\_Com

Select files using Windows Explorer. Drag them over the ListView to see a count of the files selected appearing in the "Items" column. Drop them onto the ListView to see the file paths displayed in the second column.

**Warning:** Be aware of the code in the DragOver event handler that checks the drag state. After the DragDrop completes, another DragOver event is raised, with the dragState set to **Exit**. This allows the control to perform any final processing required at the completion of the drag.

#### Also see

6.28.1 Payload
6.28.2 Example 1: Move Employees in a Tree
6.28.3 Example 2: Move Employees between Two Forms
↑ 6. Creating Applications Using Components

# 6.28.1 Payload

The object being dragged and dropped is the Payload:

Details	5
DD_EMPPL -	
E • ?	
Properties Events Methods	
Add_to_payload Cet_payload_Item	

The examples in this section use a common payload reusable part DD\_EMPPL. It contains a working list of the employee information and a method to add items to the payload and to retrieve items from it.

Source for the DD\_EMPPL Payload

↑ 6.28 Drag-and-Drop

### Source for the DD\_EMPPL Payload

Copy and paste this code to a reusable part named DD\_EMPPL and compile it: FUNCTION options(\*DIRECT) BEGIN COM role(\*EXTENDS #PRIM OBJT)

DEF LIST name(#PAYLOAD) fields(#EMPNO #SURNAME #GIVENAME =

DEFINE PTY name(Payload items) get(\*auto #listcount)

\* Add an item to the payload MTHROUTINE name(Add to payload) DEFINE\_MAP for(\*input) class(#empno) name(#Employee\_ID) mandatory(") DEFINE\_MAP for(\*input) class(#surname) name(#Employee\_Surname) manc DEFINE\_MAP for(\*input) class(#Givename) name(#Employee\_Givename) m DEFINE\_MAP for(\*input) class(#Deptment) name(#Employee\_Department) r DEFINE\_MAP for(\*input) class(#Section) name(#Employee\_Section) mandat

CHANGE field(#EMPNO) to('#Employee\_ID.value') CHANGE field(#SURNAME) to('#Employee\_Surname.value') CHANGE field(#GIVENAME) to('#Employee\_Givename.value') CHANGE field(#DEPTMENT) to('#Employee\_Department.value') CHANGE field(#SECTION) to('#Employee Section.value')

ADD\_ENTRY to\_list(#PAYLOAD)

**ENDROUTINE** 

MTHROUTINE name(Get\_payload\_Item) DEFINE\_MAP for(\*input) class(#Listentry) name(#Payload\_item) DEFINE\_MAP for(\*output) class(#empno) name(#Employee\_ID) mandatory( DEFINE\_MAP for(\*output) class(#surname) name(#Employee\_Surname) mar DEFINE\_MAP for(\*output) class(#Givename) name(#Employee\_Givename) 1 DEFINE\_MAP for(\*output) class(#Deptment) name(#Employee\_Department)

DEFINE\_MAP for(\*output) class(#Section) name(#Employee\_Section) manda

CHANGE field(#LISTENTRY) to('#PAYLOAD ITEM.VALUE') GET\_ENTRY number(#LISTENTRY) from list(#PAYLOAD)

SET com(#Employee\_ID) VALUE(#empno) SET com(#Employee\_Surname) VALUE(#surname) SET com(#Employee\_Givename) VALUE(#givename) SET com(#Employee\_Department) VALUE(#deptment) SET com(#Employee\_Section) VALUE(#section)

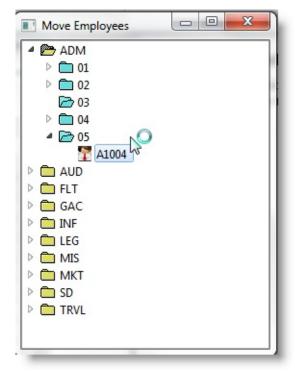
ENDROUTINE

END\_COM

↑ 6.28.1 Payload

### 6.28.2 Example 1: Move Employees in a Tree

This example contains a tree view of departments, sections and employees. You can move employees between sections by drag-and-drop.



Source for the Employee Drag-and-Drop Tree ↑ 6.28 Drag-and-Drop

### Source for the Employee Drag-and-Drop Tree

Copy and paste this code to a form and compile and execute it. Note that you must have the Source for the DD\_EMPPL Payload in a compiled reusable part for this example to work.

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #PRIM\_FORM) CAPTION('Move Employee DEFINE\_COM class(#PRIM\_ATLM) name(#ATLM\_1) DEFINE\_COM class(#PRIM\_TRVW) name(#TRVW\_1) DISPLAYPOSITION DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_2) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_TVCL) name(#TVCL\_3) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_ATLI) name(#ATLI\_1) ATTACHMENT(Center DEFINE\_COM class(#PRIM\_STBR) name(#STBR\_1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_ATLI) name(#ATLI\_4) ATTACHMENT(Botton

\* Employee Payload object

DEFINE\_COM class(#dd\_emppl) name(#PAYLOAD) reference(\*dynamic)

DEFINE field(#W\_EMPNO) reffld(#EMPNO) DEFINE field(#W\_DEPT) reffld(#DEPTMENT) DEFINE field(#W\_SECT) reffld(#SECTION)

EVTROUTINE handling(#com\_owner.CreateInstance) options(\*NOCLEARE \* Tree view images SET com(#tvcl\_1) IMAGE(#vi\_foldcl) IMAGEEXPANDED(#vi\_foldop) SET com(#tvcl\_2) IMAGE(#vi\_sectcl) IMAGEEXPANDED(#vi\_sectop) SET com(#tvcl\_3) IMAGE(#vi\_employ)

\* Populate Tree SELECT fields(#TRVW\_1) from\_file(DEPTAB) SELECT fields(#TRVW\_1) from\_file(SECTAB) with\_key(#DEPTMENT) SELECT fields(#TRVW\_1) from\_file(PSLMST1) with\_key(#DEPTMENT #S ADD\_ENTRY to\_list(#TRVW\_1) ENDSELECT ENDSELECT ENDSELECT ENDROUTINE \* Start the drag operation.
\* Only allow employees to be "dragged"
EVTROUTINE handling(#TRVW\_1.StartDrag) options(\*NOCLEARMESSA(

USE builtin(CLR\_MESSAGES)

\* Create Payload Instance SET\_REF com(#payload) to(\*create\_as #dd\_emppl)

\* Only allow employee to be dragged IF cond('#Trvw\_1.currentitem.level = 3')

\* Add item to the payload INVOKE method(#Payload.Add\_to\_payload) EMPLOYEE\_ID(#empno) EMF ELSE

SET com(#continue) VALUE(false) ENDIF

ENDROUTINE

EVTROUTINE handling(#TRVW\_1.DragOver) options(\*NOCLEARERROR \* Show Item dragged over SET com(#showhilight) VALUE(true)

\* If a department, employee cannot be dropped IF\_REF com(#trvw\_1.currentitem) is\_not(\*null)

IF cond('(#trvw\_1.currentitem.level \*ne 1)')

\* Allow drop and set new cursor SET com(#acceptdrop) VALUE(true)

ELSE

\* Disable drop SET com(#acceptdrop) VALUE(False)

#### ENDIF

#### ENDIF ENDROUTINE

EVTROUTINE handling(#TRVW\_1.DragDrop) options(\*NOCLEARERROR

\* If the payload is not empty, get the payload item and add to the treeview in th IF cond('#Payload\_Payload\_Items \*ne \*zero')

\* Return values in to working fields fields to compare with current item values INVOKE method(#Payload.Get\_Payload\_Item) PAYLOAD\_ITEM(1) EMPLC

\* If the department and/or section have changed, do the drop IF cond('(#deptment \*ne #W\_dept) or (#section \*ne #w\_sect)')

\* Department and section values of currentitem are used ADD\_ENTRY to\_list(#TRVW\_1)

MESSAGE msgtxt(' Drop Successful')

\* Expand Parent items of new drop point

\* As drop can only happen on a section, department must be expanded

CHANGE field(#W\_DEPT) to(#DEPTMENT) CHANGE field(#W\_SECT) to(#section)

SELECTLIST named(#TRVW\_1)

CONTINUE if('#w\_dept \*ne #deptment') CONTINUE if('#w\_sect \*ne #section')

SET com(#trvw\_1.currentitem) EXPANDED(true)

LEAVE ENDSELECT

ENDIF

#### ENDIF

#### ENDROUTINE

EVTROUTINE handling(#TRVW\_1.EndDrag) options(\*NOCLEARERRORS

IF cond('#Dragresult.value = Accepted')

\* If the department and/or section have changed delete the source employee IF cond('(#deptment \*ne #w\_dept) or (#section \*ne #w\_sect)')
\* Get payload item details INVOKE method(#Payload.Get\_Payload\_Item) PAYLOAD\_ITEM(1) EMPLC

\* Locate and Delete source employee SELECTLIST named(#TRVW\_1) CONTINUE if('#w\_dept \*ne #deptment') CONTINUE if('#w\_sect \*ne #section') CONTINUE if('#w\_empno \*ne #empno')

DLT\_ENTRY from\_list(#TRVW\_1)

LEAVE ENDSELECT

ENDIF ENDIF ENDROUTINE END\_COM

↑ 6.28.2 Example 1: Move Employees in a Tree

# 6.28.3 Example 2: Move Employees between Two Forms

In this example employees are dragged from one form and dropped to another form.

This is the source form:

Employee Num	Employee Surname	Employee Given Na	Department Code	Section Code	
A0070	Brown	Veronica	INF	DV	
A0193	Smithson	Fred	ADM	05	
A0907	Simpson	Anne	AUD	03	
A1001	Jones	Shirley	ADM	01	
A1002	Smythe	John	ADM	02	
A1004	Smithson	Ruth	ADM	03	
A1005	Smiths	Peter	ADM	02	
A1006	Smithers	Jack	TRVL	03	
41007	Snell	George	AUD	01	
A1008	Sneddon	Alison	AUD	01	
A1009	Snashall	Damian	AUD	02	-

#### And this is the target form:

Employee Num	Employee Surname	Employee Given Na	Department Code	Section Code
40090	Bloggs	Fred	FLT	03
41003	Smithe	Robert	FLT	02

If the employees are dragged without holding the Ctrl key down, they are moved from the source to the target form. If the Ctrl key is held down, the employees are copied from the source form to the target form.

#### Cursors

To visually differentiate between the move and copy operations, this example users two cursor components. Therefore if you want to execute this example you need to create two components of type Cursor:

DD\_CURMOV This cursor will be displayed when you copy employees

DD\_CURCPY This cursor will be displayed when you move employees

When you create these components, you need to specify the name of an existing .cur file as the value of the FileName property. You should be able to find .cur files in your Windows directory. It does not matter what cursor image you choose.

To execute this example, you need Source for the DD\_EMPPL Payload compiled in a reusable part, the two cursors mentioned in the previous section and the compiled Source for the Source Form and Source for the Target Form . Execute the source form. The target form will be displayed automatically.

### Source for the Source Form

Copy and paste this code to a form and compile and execute it:

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #PRIM\_FORM) HEIGHT(258) LAYOUTM/ DEFINE\_COM class(#PRIM\_ATLM) name(#ATLM\_1) DEFINE\_COM class(#PRIM\_STBR) name(#STBR\_1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_ATLI) name(#ATLI\_1) ATTACHMENT(Botton DEFINE\_COM class(#PRIM\_LTVW) name(#LTVW\_1) DISPLAYPOSITION DEFINE\_COM class(#PRIM\_ATLI) name(#ATLI\_2) ATTACHMENT(Center DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_2) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_3) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_4) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_5) DISPLAYPOSITION(

DEFINE\_COM class(#DD\_TARGET) name(#DD\_TARGET)

EVTROUTINE handling(#com\_owner.CreateInstance) SET com(#com\_owner) CAPTION(\*component\_desc) SELECT fields(#LTVW\_1) from\_file(PSLMST) ADD\_ENTRY to\_list(#LTVW\_1) ENDSELECT INVOKE method(#dd\_target.showform) INVOKE method(#com\_owner.activateform) ENDROUTINE

EVTROUTINE handling(#LTVW\_1.StartDrag) PAYLOAD(#Payload) DEFINE\_COM class(#dd\_emppl) name(#Local\_Payload)

\* Create payload SELECTLIST named(#LTVW\_1) CONTINUE if('#ltvw\_1.currentitem.selected = False') INVOKE method(#Local\_Payload.Add\_to\_payload) EMPLOYEE\_ID(#empn( ENDSELECT

\* Move temporary payload to Drag Manager.\* Temporary Payload is destroyed at the end of the routine

SET\_REF com(#payload) to(#Local\_Payload)

ENDROUTINE

\* Check Drag result EVTROUTINE handling(#LTVW\_1.EndDrag) PAYLOAD(#payload) DRAGF DEFINE\_COM class(#dd\_emppl) name(#Local\_Payload) reference(\*dynamic

SET\_REF com(#Local\_payload) to(\*dynamic #Payload)

CHANGE field(#STD\_OBJ) to('#DRAGRESULT.VALUE')

IF cond('#DragResult.value = Accepted')

\* Control key down then copying IF cond('#sys\_keybd.ControlkeyDown = False')

SELECTLIST named(#LTVW\_1) CONTINUE if('#ltvw\_1.currentitem.selected = False') DLT\_ENTRY from\_list(#LTVW\_1) ENDSELECT ENDIF

ENDIF

ENDROUTINE

EVTROUTINE handling(#COM\_OWNER.Closing) options(\*NOCLEARMES INVOKE method(#dd\_target.closeform) ENDROUTINE END\_COM

↑ ↑ 6.28.3 Example 2: Move Employees between Two Forms

# Source for the Target Form

Copy and paste this code to a form and compile it. This form is displayed automatically with the source form.

FUNCTION options(\*DIRECT) BEGIN\_COM role(\*EXTENDS #PRIM\_FORM) HEIGHT(256) LAYOUTM/ DEFINE\_COM class(#PRIM\_ATLM) name(#ATLM\_1) DEFINE\_COM class(#PRIM\_STBR) name(#STBR\_1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_ATLI) name(#ATLI\_1) ATTACHMENT(Botton DEFINE\_COM class(#PRIM\_LTVW) name(#LTVW\_1) DISPLAYPOSITION DEFINE\_COM class(#PRIM\_ATLI) name(#ATLI\_2) ATTACHMENT(Center DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_1) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_2) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_3) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_4) DISPLAYPOSITION( DEFINE\_COM class(#PRIM\_LVCL) name(#LVCL\_5) DISPLAYPOSITION(

EVTROUTINE handling(#com\_owner.CreateInstance) SET com(#com\_owner) CAPTION(\*component\_desc) ENDROUTINE

EVTROUTINE handling(#ltvw\_1.DragOver) PAYLOAD(#payload) ACCEPT \* If an instance of the Employee "Payload" SET com(#Acceptdrop) VALUE(true)

IF cond('#sys\_keybd.ControlkeyDown = False')

SET\_REF com(#cursor) to(#dd\_curmov)

ELSE

SET\_REF com(#cursor) to(#dd\_curcpy)

ENDIF ENDROUTINE

EVTROUTINE handling(#ltvw\_1.DragDrop) PAYLOAD(#payload) DEFINE\_COM class(#dd\_emppl) name(#Local\_Payload) reference(\*dynamic SET\_REF com(#Local\_Payload) to(\*dynamic #Payload)

\* Get the number of items from the payload

CHANGE field(#STD\_NUM) to('#Local\_Payload.Payload\_Items')

\* Get the number of items data from the payload

BEGIN\_LOOP using(#LISTENTRY) to(#STD\_NUM) INVOKE method(#Local\_Payload.Get\_Payload\_Item) PAYLOAD\_ITEM(#lis ADD\_ENTRY to\_list(#LTVW\_1) END\_LOOP

SET\_REF com(#Local\_Payload) to(\*null)

ENDROUTINE END\_COM

↑ ↑ 6.28.3 Example 2: Move Employees between Two Forms

## 6.29 What the Application will look like

In Visual LANSA you define the way the application will be displayed using visual styles and layout managers.

A visual style is a component which sets fonts, colors and other display attributes. A layout manager is a component which defines how components are positioned on a form, panel, tab sheet or other container.

6.29.1 Layout Managers

6.29.2 DirectX Styles

6.29.3 Visual Styles

6.29.4 Themes

1 6. Creating Applications Using Components

## 6.29.1 Layout Managers

There are two ways you can specify how components are placed on a form, panel, tab sheet or other container:

- By positioning them manually in the editor
- By using a layout manager

Positioning components manually is easier at design time. However, if the application is minimized, resized or maximized at run time, the components in it are not resized or repositioned. Also, organizing components in a particular layout (aligned and evenly spaced) can be tricky.

Layout managers control how components are laid out inside a container. At design time their use can first seem a bit complex, but they add considerable value to the appearance, usability and maintainability of your applications.

New     Rows     Height     1       Layout     Row_1	<ul> <li>♦ Width</li> <li>Fixed Width</li> <li>Columns</li> <li>Fixed Width</li> <li>Column_1</li> </ul>
New Layout	Layout Manager - DEMCOM01 🕞
Table   Table   Table Layout Manager   Table Layout Manager     Table Layout Manager     Table Layout Manager     Table Layout Manager     Table Layout Manager     Table Layout Manager     Flow Down   Flow Down Layout Manager   Flow Across   Flow Across Layout Manager   Split Horizontal   Split Horizontal Layout Manager   Split Vertical   Split Vertical	

Which Components Have Layout Managers?

All components that can contain other components have a layout manager. The most obvious ones are forms and reusable parts.

Typically you would define layout managers also for group boxes and tab sheets.

The Table Layout Manager is the most commonly used layout manager.

#### Add Components to the Layout

Any components you add to a container that has a layout manager are automatically attached to the layout.

#### See also

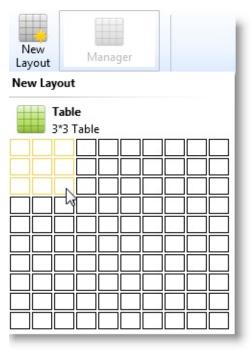
Table Layout ManagerOther Layout Managers↑ 6.29 What the Application will look like

#### **Table Layout Manager**

The table layout is the best layout to use in most situations.


A Table layout divides the container into rows and columns.

You choose the number of columns and rows when creating the layout:



#### Also see

Understand the Table Layout Create a Table Layout Add Controls to a Table Layout Table Layout Manager

## **Understand the Table Layout**

Table Layout divides the space into rows and columns (and dividers) which can be a fixed pixel width/height or a proportion of the available space.

When you drop a control into a layout (or when you start to specify layout features for a control), a layout item is created to manage the control's positioning. The layout item has properties which define an imaginary rectangle inside which the control is positioned: Row, Column, Row Span and Column Span. The layout item's Sizing, Alignment, MarginXxx, and Flow properties position the control within the rectangle.

The Alignment property can put the control in one of nine possible positions, such as TopLeft, Center, BottomRight etc. Because the layout allows controls to occupy the same space, multiple controls with the same alignment will be shown on top of one another unless the Flow property is specified.

The Sizing property determines how big the control is. Controls can fit to the width and height of the rectangle, or simply appear as designed. If a row or column resizes, the control is also resized.

The Flow property is used to force controls to appear in sequence to the Left, Right, Top or Bottom of other controls in the same rectangle (cell) with the same Alignment and Flow. In this way you can have different groups of controls with different flows occupying the same cell. It is possible that a control will appear outside of its specified row and column due to Flow behaviour, but it will still size and align as though it was in the cell.

Except for the Flow property, the layout item's properties are independent of each other. This means that if you change the row and column span of a control to make it wider, the remaining controls will not be affected.

↑ Table Layout Manager

## **Create a Table Layout**

To add a Table layout to a component, use the New Layout command on the Design tab of ribbon and choose the number of rows and columns required. Afterwards you can also add vertical and horizontal dividers to the layout if required:

AaBb Style Private Mouseover Style	New ATLM_1
Style	New Layout
Design Source Repository Details Repository H	
	3*2 Table
	Attachment Attachment
	Flow Down
	Flow Down Layout Manager
	Flow Across Layout Manager
	Split Horizontal Layout Manager
	Split Vertical Split Vertical Layout Manager

The layout is created:

aBb /	ite Mouseover	New New Style Layout	Rows Fixed Heig	ht 📃 🛛 🔸 Row_1	Columns	Fixed Width 🔲 Column_1
Style				Layo	ut Manager - D	EMCOM01
Source Ren	ository Details Re	pository Help   Cross	References			
		P = = = = = = = = = = = = = = = = = = =				
					6	
						**-*-*-*-*-*-*-*-*

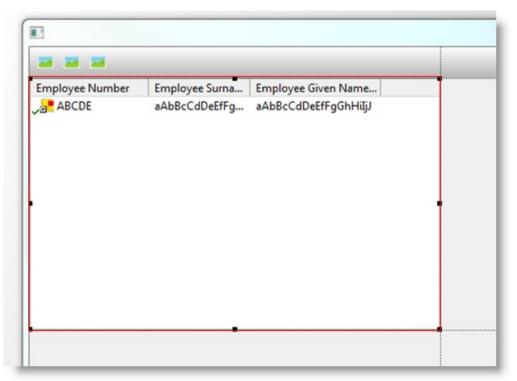
You can now start dropping controls from the Controls View or the Repository Browser onto the layout.

1 Table Layout Manager

# Add Controls to a Table Layout

When you drop controls from the Controls View or the Repository Browser onto the layout, they adopt default layout behaviour depending on their type:

• Composite controls such as a panel, lists, and user designed controls resize themselves to fit the cell:



• Labels, spin edits, check boxes and other user input controls are aligned to top left with Flow(Down) ensuring that they stack neatly down the page. The DisplayPosition property determines the order in which the controls are placed:

Employee Number	ABCDE
Employee Surname	aAbBcCdDeEfFgGhHiIjJ
Employee Given Name(s)	aAbBcCdDeEfFgGhHiIjJ
Street No and Name	aAbBcCdDeEfFgGhHiIjJkKILm
Suburb or Town	aAbBcCdDeEfFgGhHiljJkKILm
state and Country	aAbBcCdDeEfFgGhHiIjJkKILm
ost / Zip Code	123456
Home Phone Number	ABCDEFGHIJKLMNO
Business Phone Number	ABCDEFGHIJKLMNO
start date (YYMMDD)	12/34/56
Termination Date (YYMMDI	D) 12/34/56

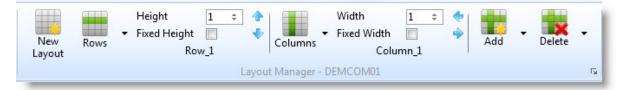
- Buttons automatically flow left when added and have an Alignment of TopLeft.
- A toolbar will show several buttons. When a toolbar button is added to the layout, it defaults to Flow(Left) so that the buttons line up correctly:

				_	
Employee Number	Employee Surna	Employee Given Name	Employee Number	ABCDE	
ABCDE	aAbBcCdDeEfFg	aAbBcCdDeEfFgGhHiljJ	Employee Surname	aAbBcCdDeEfFqGhHiIjJ	

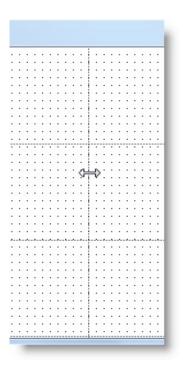
When controls are moved from one cell to another, they take their behaviour with them and simply apply it to the bounds defined by the new cell. If that happens to coincide with other controls, they are not affected.

Once added to the design all of the properties of the layout items can be modified as required, and controls can be moved and resized in the Design view.

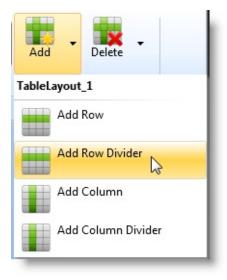
Table layout features can be accessed directly from the ribbon:



Column dividers are automatically added between the first two available divisions:



By default they occupy the space available and will always be on top. However, they can be adjusted to start and end on whichever division is required. Divisions can be added and deleted:



Controls can be made to span more than one cell. For example, here the bottom corner of the list has been dragged over the row below so it occupies rows 2 and 3:

Employee Number	Employee Surna	Employee Given Name	Employee N
ABCDE	aAbBcCdDeEfFg	aAbBcCdDeEfFgGhHiījJ	Employee S Employee ( Street No a Suburb or 1 State and C Post / Zip ( Home Pho Business Pl Start date ( Terminatio

↑ Table Layout Manager

#### **Other Layout Managers**

In the majority of cases the best way to create a layout is by using a table layout manager. However, other layout managers are available for backward compatibility and for special circumstances.

These layout managers are:

Attachment Layout Manager Flow Layout Manager Grid Layout Manager Split Layout Manager Specify the Layout Manager Set the Layout Properties Run Applications with Layout Managers  $\uparrow$  6.29 What the Application will look like

## **Attachment Layout Manager**



An attachment layout divides the container into five cells. The center cell expands to fill any available space.

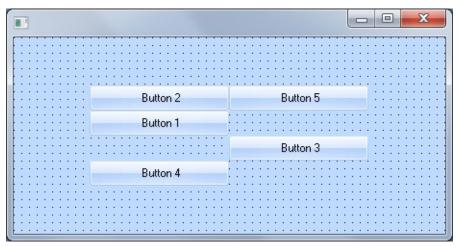
#### **Flow Layout Manager**

Button 5
Button 4
Button 3
Button 2
Button 1

The Flow Layout Manager is used to position components in a display area by dividing the items into rows or columns and then flowing the items one after the other in the appropriate row/column.

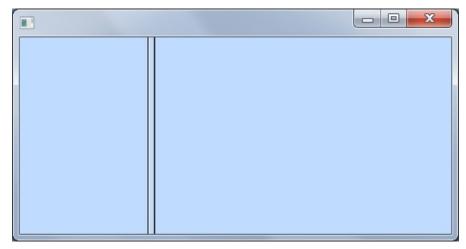
Typically the item are flown either from top to bottom (Flow Down) or from right to left (Flow Across)

## **Grid Layout Manager**



The Grid Layout Manager is used to position components in a display area that corresponds to a grid (table). The components are placed in specific 'cells' of the layout by specifying the column from left and the row from top.

## Split Layout Manager



The Split Layout Manager is used to manage a component's display area so that the component's children are split either across or down the display area.

The split layouts can be horizontal (Split Horizontally) or vertical (Split Vertically).

## Specify the Layout Manager

You specify the layout manager for a container using its LayoutManager property. By default this property is set to \*NULL.

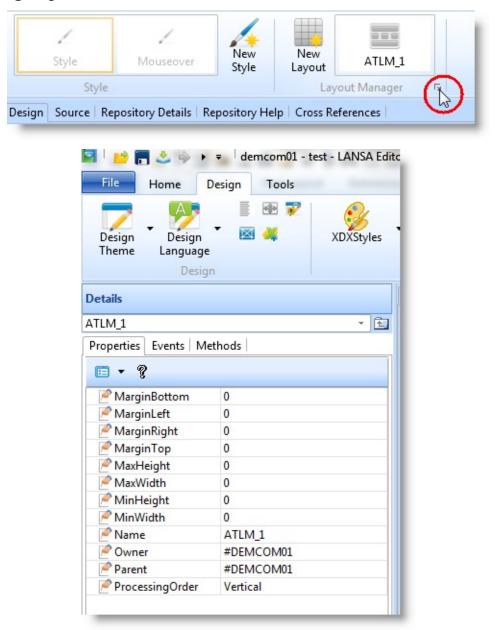
To choose a layout manager use the Layouts group in the Design tab in the ribbon.

Nev Layout	Rows	Height Fixed Height Ro	1 ow_1		
New Lay	out		- 1		
Table Table Layout Manager					
	Attachment Clow Down Clow Across Clow Across La Split Horizon	yout Manager	er		
Split Horizontal Layout Manager Split Vertical Split Vertical Layout Manager					

Then lay out individual components using the layout options on the ribbon:



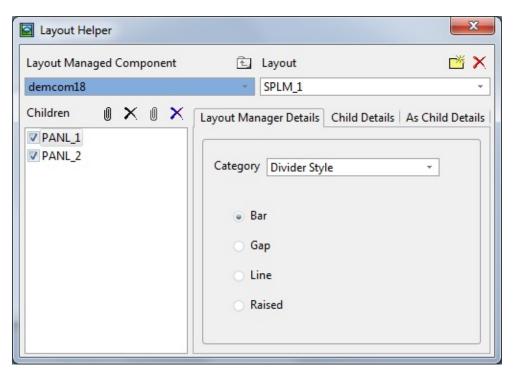
You can also display the layout details by using the arrow on the Layout Manager group in the ribbon:



Or click on the button with ellipses (three dots) in the LayoutManager property of the Details tab to display the Layout Helper:

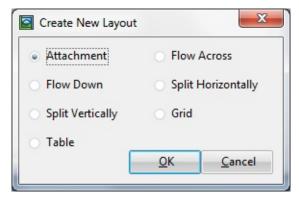
ImageAlignment	Center	=
LayoutManager	*NULL	- ·;
🖻 Left	854	3

The Layout Helper dialog is displayed:



Use the Layout Managed Component drop-down to choose the component for which you specify a layout.

When you have selected the component, click on the Create layout manager button to select the type of layout you want to create. The Create New Layout dialog is displayed:



Choose a layout and click OK.

The new layout (in this case ATLM\_1) is displayed in the Layout field:

Layout Helper			×
Layout Managed C	omponent	主 Layout	🛎 🗙
demcom18		<ul> <li>ATLM_1</li> </ul>	•
Children 🕖	× • ×	Layout Manager Details Child Details As Chi	d Details
PHBN_1 PHBN_2 PHBN_3 PHBN_4		Category Processing Order -	
PHBN_5		<ul> <li>Vertical</li> </ul>	
		<ul> <li>Horizontal</li> </ul>	
		<ul> <li>Display Position</li> </ul>	

All the components contained in the layout managed component are shown in the list of Children.

Any components you add to the layout managed component after the layout manager has been specified will automatically be attached to the layout.

To view the Layout Helper again click on the button with ellipses in the LayoutManager property in the Details tab.

## **Set the Layout Properties**

You may want to adjust the properties of the layout. To do this you can set the properties of the layout manager in the Details tab. However, the easiest way to work with the layout managers is to use the Layout Helper.

To display the Layout Helper use the Layouts tab in the ribbon. Choose the component you want to work with from the Layout Managed Component drop-down. The layout specified for it is displayed in the Layout drop-down.

Layout Helper		x
Layout Managed Component	🚉 Layout	🔺 🗙
demcom18	- ATLM_1	
Children U X U X PHBN_1 PHBN_2 PHBN_3 PHBN_4 PHBN_5	Layout Manager Details Child Details As Ch Category Processing Order	ild Details

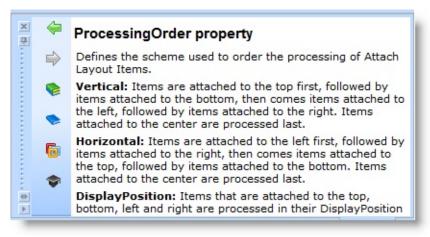
Use the Layout Manager Details tab to set the properties of the layout .

Make sure that you have child components attached to the layout, otherwise you will not see the effects of the changes you make. Then choose the type of properties you want to work with from the Category drop-down:

Layout Helper	X
Layout Managed Component	Layout 🛃 🗙
Children () X () X PHBN_1 PHBN_2 PHBN_3 PHBN_4 PHBN_5	Layout Manager Details Child Details As Child Details          Category       Processing Order         Processing Order       Processing Order         Margins       Ver Minimum         Horizontal       Display Position

Display the component in the Design view and make any changes you want to the individual properties of the layout in the Layout Helper. These changes are immediately reflected in the layout managed component.

If you want a detailed description of a particular property, select it and press F1 to bring up its Feature help:



Also see

Set the Properties of the Child Components Display the Layout Manager The Layout Manager Definitions Attachment Layout Split Layout Grid Layout Flow Layout ↑ Other Layout Managers

## **Run Applications with Layout Managers**

At run-time when you resize the form, the components stay attached to the positions specified in the layout:

Ī	Button 1	1
Button 4	Button 5	Button 2
1	Button 3	ļ
	Button 1	
Button 4	Button 5	Button 2
	Button 3	4

1 6.29 What the Application will look like

## 6.29.2 DirectX Styles

DirectX Styles are not available in Win32 runtime.

Evolving web page designs, mobile phone apps, large monitors, and the rise of the touch screen pose challenges in the design of user interfaces. DirectX styles offer you flexibility in creating your application user interfaces and provide effects such as transparency, rounded corners, gradient brushes and flat designs.

Style Private	Mouseover Style		Manager		
Local Styles					
1					
*NULL					
XDXStyles - Direct	X Styles				
AaBb	AaBb	AB	AaBb	AaBb	*
Background	CaroActiveItemBG	Centered	Combo	ContentBackground	
AaBb	AaBb	AaBb	AaBb	AaBb	
DragOver	Edit	Error	Found	Heading	
AaBb	AaBb	AaBb	AaBb	AaBb	
Hint	HintArrow	Item	ItemHeading	Label	
AaRh	<b>DK</b>	AaBb	AaBb	AaBb	
LargeFont	LargeFonts	LargeHeading	MouseOver	MouseOverBorder	
AaBb	AaBb	AaBb	AaBb	AaBb	
Popup	PopupButton	SpinEdit	Transparent	Tree	-

The DirectX Style property controls the way components are displayed when run in DirectX mode:

i • ?		
🖉 ImageAlignment	Center	
🖻 LayoutManager	#ATLM_1	
neft 🖉	444	
nenuBar 🖉	*NULL	
MouseOverStyle	*NULL	
name 🖉	DEMCOM01	
ner 🖉	#DEMCOM01	
nerent 🖉	#DEMCOM01	
🦻 Popup	*NULL	
ne PopupMenu 🖉	*NULL	
PrivateStyle	*NULL	
RenderStyle	Application	ſ
ScalingFactorHor	9600	
ScalingFactorVer	9600	
🖻 ShowIn Taskbar	FormStyle	
🖉 Style	#XDXStyles <background></background>	
🖉 Taskbar	*NULL	
ThemeDrawStyle	None	
ThemeStyle	Application	:
TitleBarVisible	True	
🖻 Тор	235	
Visible	True	
VisualStyle	*NULL	
VisualStyleOfParent	True	
🔎 Width	692	
WindowState	Normal	

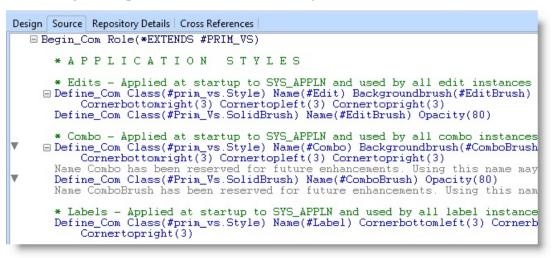
#### Also see

Defining Styles Using Styles MouseOverStyle and MouseOverStyles Properties PrivateStyle and PrivateStyles Properties 1 6.29 What the Application will look like

## **Defining Styles**

DirectX Styles are defined as features of a Visual Style. They can be thought of as much the same as cascading style sheet, with multiple styles defined in one place.

The following example shows a few basic styles:



Unlike Visual Styles, DirectX Styles only affect the features specifically referenced in their definition.

DirectX Styles are made up of background and foreground features. Only the foreground features - TextColor, Font and ForegroundBrush - are adopted by child controls. This is because borders and a background brush applied to a panel are usually not required on child controls. Instead, panels and labels are transparent by default, allowing a base background to be visible. If you want the background and border to repeat on each control, you can explicitly apply the Style.

## **Using Styles**

All controls have Style and Styles properties. The simplest application of a style follows:

```
Define_Com Class(#Prim_labl) name(#Heading) Caption('Details')
Style(#MyStyles<Heading1>)
```

Multiple styles can be assigned to a single control at runtime by using the Styles property.

In this example, a label already has a style, but as a value changes, an 'error' style is added or removed as required:

```
Define_Com Class(#Prim_labl) name(#ValueCaption) Caption('Details')
Style(#MyStyles<Emphasis>)
Evtroutine Handling(#Value.Changed)
If (#Value > #AllowedLimit)
#ValueCaption.Styles.Add(#MyStyles<Error>)
Else
#ValueCaption.Styles.Remove(#MyStyles<Error>)
Endif
Endroutine
```

Styles can also be applied at a global level, for example, when the application first starts. The styles are adopted by all forms and reusable parts.

This same concept also applies to individual controls. Following, the application is given a style, and an additional style is applied to all edit boxes to ensure that they all have rounded corners when first created:

```
Evtroutine Handling(#Com_owner.CreateInstance)
#sys_appln.Style <= #MyStyles<ApplicationStyle>
#sys_appln.Appearance.Edit <= #MyStyles<Edit>
Endroutine
```

#### MouseOverStyle and MouseOverStyles Properties

Applications often change the appearance of controls as the mouse enters. To simplify this process, all controls have a further pair of properties: MouseOverStyle and MouseOverStyles.

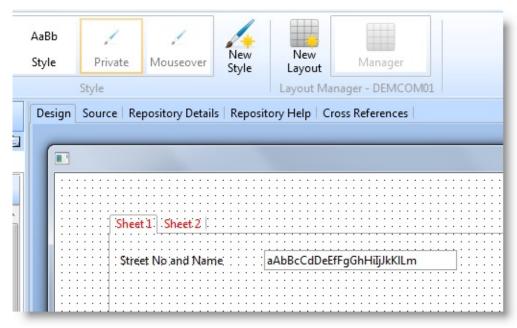
As with Style and Styles, the control can be defined as having an individual MouseOverStyle, or multiple MouseOverStyles can be applied at runtime.

Define\_Com Class(#Prim\_labl) name(#ValueCaption) Caption('Details')
MouseOverStyle(#MyStyles<Emphasis>)

The Visual LANSA runtime automatically adds the style when the mouse enters the control and removes it again as soon as the mouse leaves. This saves you cluttering the code with multiple MouseEnter and MouseLeave routines.

## **PrivateStyle and PrivateStyles Properties**

Composite controls, such as tab folder and group box often need their own style for captions, without the need for that style to be inherited by their child controls. To facilitate this, composite controls can have their own private styles. These are styles that are applied to the visible features of the control, typically text, but will not be adopted by the child controls.



## **Runtime Styles**

For most scenarios, predefined styles stored in the repository will cover application requirements. However, there are situations where dynamic styles are required, for example for font or color selection. In these situations, Styles can be defined and instances created just like any other component, meaning their properties can be set at runtime.

The following shows a color picker dialog when the form is clicked. The selected color is applied to the style and thus to the background of the form:

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_Form) Clientheight(284)
Clientwidth(464) Height(322) Left(126) Style(#Style) Top(223) Width(480)
Define_Com Class(#prim_vs.Style) Name(#Style) Normbackcolor(White)
Define_Com Class(#Prim_appl.ICommonDialogColor) Name(#ColorPicker)
Reference(*dynamic)
Evtroutine Handling(#Com_owner.CreateInstance)
#Com_owner.Cursor <= #sys_appln.Cursors<Hand>
Endroutine
Evtroutine Handling(#Com_owner.Click)
Define_Com Class(#prim_boln) Name(#OK)
If (#ColorPicker *Is *null)
#ColorPicker <= #sys_appln.CreateColorDialog
#ColorPicker.FullyOpen := true
Endif
```

### 6.29.3 Visual Styles

To define how, for example, a label will be displayed, you set its VisualStyle property.

🖉 TitleBarVisible	True	
🦻 Тор	108	-
🖻 Visible	True	
VisualStyle	#VS_NORM	
VisualStyleOfPai	True	
P Width	662	
WindowState	Normal	

Let's say you use the shipped visual style VS\_Norm. This style defines the following properties for the label:

- BorderStyle 3DLeft (3D effect, shadow on the left)
- NormBackColor ButtonFace (the color specified for buttons in the Windows BUTTONFACE system value, usually gray)
- TextColor WindowText (the color specified for windows text in the WINDOWTEXT system value, usually black)
- Face Name MS Sans Serif
- FontSize 8 points

It will look like this:

Make sure you fill in all the fields.

You could, for instance, change the VisualStyle of you label to VS\_WARN. This style has different TextColor and Font properties:

TextColor: RedBold: True

If you used this style, your label would be displayed like this:

Make sure you fill in all the fields.

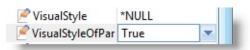
#### Also sefurtherf

The Scope of Visual Styles Shipped Visual Styles Set the Default Visual Style for New Forms Create or Edit a Visual Style Set Colors Preview your Visual Style ↑ 6.29 What the Application will look like

## The Scope of Visual Styles

There are two ways of setting a visual style: using the VisualStyle property or the VisualStyleOfParent property:

- The VisualStyle property sets the style of the individual component to a style. It's value is the name of a style, for example VS\_Norm.
- The VisualStyleOfParent specifies whether this component is to use the visual style of its parent. Its value can be True or False.



By default, the VisualStyle of components is set to \*NULL and their VisualStyleOfParent is set to True. This means that the components use whatever style has been assigned to their parent component. Typically the parent component is the form.

In most circumstances it is best that you do not change this default. This is because if you assign a visual style to your form and keep the VisualStyleOfParent of all the other components set to True, you can change the style of all the components on the form simply by changing one value (the value of the form's VisualStyle property).

As an example, you might want to change the font size of the text the whole application. Let's say your form is using visual style VS\_Norm which specifies an 8-point font:

Employee Surname	aAbBcCdDeEfFgGhHiljJ ::;
Employee Given Name(s)	aAbBcCdDeEfFgGhHiljJ
Street No and Name	aAbBcCdDeEfFgGhHiljJkKILm
Suburb or Town	aAbBcCdDeEfFgGhHiljJkKILm
State and Country	aAbBcCdDeEfFgGhHiljJkKILm
Post / Zip Code	123456
	Save

The easiest way to change the font size to 12 points for all the components on this form would be to assign the visual style VS\_Large as the VisualStyle property of the form. Provided that you had kept the VisualStyleOfParent set to

True in the other components, they would automatically inherit this style:

Employee Surname	aAbBcCdDeEfFqG	ihHiljJ :::	
Employee Given Nam	aAbBcCdDeEfFgG	ihHiljJ	
Street No and Name	aAbBcCdDeEfFgG	ahHiljJkKILm	
Suburb or Town	aAbBcCdDeEfFgG	ihHiljJkKILm 💠	
State and Country	aAbBcCdDeEfFgG	ihHiljJkKILm	
Post / Zip Code	123456		
		Save	

Of course there are cases when you want to change the visual style of an individual component such as for a label, for example. In these cases you set the VisualStyleOfParent to False.

Styles can be set at design time or when the program is running. To change a visual style at run-time you specify:

SET COM(#MY\_LABEL) VisualStyle('#VS\_WARN')

## **Shipped Visual Styles**

LANSA comes with some predefined visual styles. You can use them, change them or create your own. Note however that to get the full benefit of visual styles they should be managed centrally.

### Set the Default Visual Style for New Forms

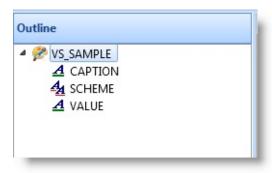
By default all components, including forms, have their VisualStyle property set to \*NULL. If you are using one style consistently in all your applications, you may want to make that style the default for all the new forms you create.

To do this, edit the standard form definition, STD\_FORM. Locate it in the Fields/Components list and open its definition. In the Properties sheet, specify the name of the visual style you want to use. Save the definition. From now on all new forms you create will use this visual style as the default.

## Create or Edit a Visual Style

To create a new visual style, use the New option of the File menu.

The editor opens the visual style. If you double-click the visual style in the Outline tab, you can see its components.



A visual style consists of a schemes and style items. By default every visual style has one Scheme and two visual style items: Caption and Value. The scheme maps Caption and Value objects to the appropriate styles. Normally you do not need to modify the scheme at all.

The Caption style item defines how controls such as labels are displayed. In the VS\_Norm style this style item has background color set to the system color ButtonFace which on most workstations is gray, the font to MS Sans Serif 8 points and the text color to Windows text color which is usually black. The border style is set to 3D Left which produces a 3D effect with shadows on the left.

The Caption style item governs these component types:

- check box
- field label
- grid headings
- group box
- list view headings
- push button
- radio button
- label
- status bar
- tab folder
- tab sheet

• track bar

The Value style item defines how controls such as fields are displayed. In the VS\_Norm style this style item is similar to the Caption style item except it uses the system color for Window (usually white) as the background color. It also defines an alternate background color for every second line of a list and an error color that can be used in case the value entered is incorrect.

The Value style item governs these component types:

- combo box
- edit box
- field edit box
- grid cells
- list box
- list view
- spin edit
- tree view

A field is a good example of the different purposes of the two styles. The way the label of the field is displayed is determined by the Caption style item, and the way the text portion of the field is displayed is governed by the Value style item:

Employee Surname	aAbBcCdDeEfFgGhHiliJ
· · · · · · · · · · · · · · · · · · ·	

The properties of a style item are set using the property sheet:

APTION	
🗆 <b>•</b> ?	
roperties Events	Methods
🖉 AlternBackColo	r Tooltip
ne Bold	False
BorderColor	3DMediumShadow
🖻 BorderStyle	None
ComponentClas	ssl PRIM_VSI
ComponentPatt	ei PRIM_VSI
ComponentTag	
ComponentTyp	el PRIM_VSI
ErrorBackColor	Error
FaceName	
FontSize	0
🔎 Italic	True
LineColor	Black
🖉 Name	CAPTION
NormBackColor	ButtonFace
Owner	#VS_SAMPLE
Parent	#VS_SAMPLE
Script	Default
SQLNullBackCo	lo SQLNull
StrikeOut	False
TextColor	WindowText
P Underline	False

Refer to the Features help text for more information about the properties of visual style items.

Note that the AlternBackColor, BorderColor and ErrorBackColor only have effect on component types (mainly lists and fields) which are governed by the Value style item. This is because AlternBackColor only applies to lists, ErrorBackColor to lists and the edit portion of a field, and BorderColor to grid lines.

## Set Colors

Color attributes can be specified in three ways:

- Standard color names such as 'Red' or 'Blue'
- As mappings to Windows system colors, such as WindowText. Many of the system colors are set using the Appearance tab of the Display object in the Control panel.
- Entering an RGB value in the format RRR:GGG:BBB. For example in VS\_Norm style the error background color in the Value item is specified as 255:96:160 (pink).

Note also that the styles shipped with LANSA deliberately avoid using hardcoded standard colors and instead use the system colors of Windows objects. In this way the Windows settings of a workstation are used in the application. Often this is the best approach because LANSA visual styles do not control the color of all Window objects (such as title bars, scroll bars etc.) and therefore it is hard to be sure that a hardcoded color would suit the chosen Windows color scheme on every workstation.

# **Preview your Visual Style**

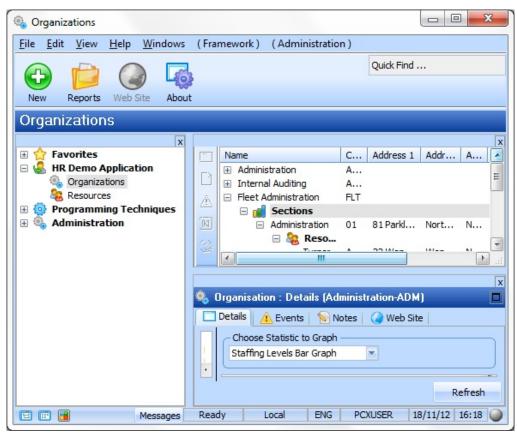
When you are editing a Visual Style, the Design tab shows a few sample components which help you visualize what your style will look like.

If you have a form open at the same time as you are editing a visual style, the changes you make to the style will only be reflected in the form when you save the style.

### 6.29.4 Themes

You can execute Visual LANSA applications using different color themes such as Office 2007 Silver, Graphite, Olive or Blue or one of the Office 2003 styles.

This picture shows the Visual LANSA Framework application executing with the Office2007 Blue theme:



#### Also see

Using Themes

Setting the Theme of All Forms

Available Themes

Theme Properties of Controls

Customizable 2003Color Theme Type

Themes and VisualStyles

1 6.29 What the Application will look like

## **Using Themes**

Visual LANSA Themes are enabled through the application object Sys\_Appln. Usually you enable a theme like this:

1. Set the theme for the application in its startup code (typically in the CreateInstance event of application main form):

#Sys\_Appln.Theme := 2007Blue

2. Enable themes for the main form of the application. This can be done from the property sheet for the form or programmatically:

#Main\_Form.ThemeStyle := Themed

Compiling and running your application will now paint your main form as a Themed form.

	-			c	
	Button 1			Button 1	
Button 4	Button 5	Button 2	Button 4	Button 5	Button 2
	Button 3			Button 3	

#### Setting the Theme of All Forms

Instead of individually setting each form to be theme-enabled using the forms' ThemeStyle=Themed property, you can set the default behavior of all Forms using the ThemedForms property of #Sys\_Appln.

False Default

True All forms in the application will use the theme specified by #Sys\_Appln.Theme property.

To do this place the following code in the application startup code:

#Sys\_Appln.Theme := Shell
#Sys\_Appln.ThemedForms := True

Note that enabling themes in all forms in a Visual LANSA application may not produce the appearance that you want.

For example, in the Microsoft Office applications, only the main form is generally theme-enabled whereas all other dialogs are not (the main application appears with a Blue theme but its child dialogs still appear Grey). For this reason you should individually evaluate forms to see if they should use a theme.

#### **Available Themes**

The Theme property of the Sys\_AppIn object sets the theme of the application:

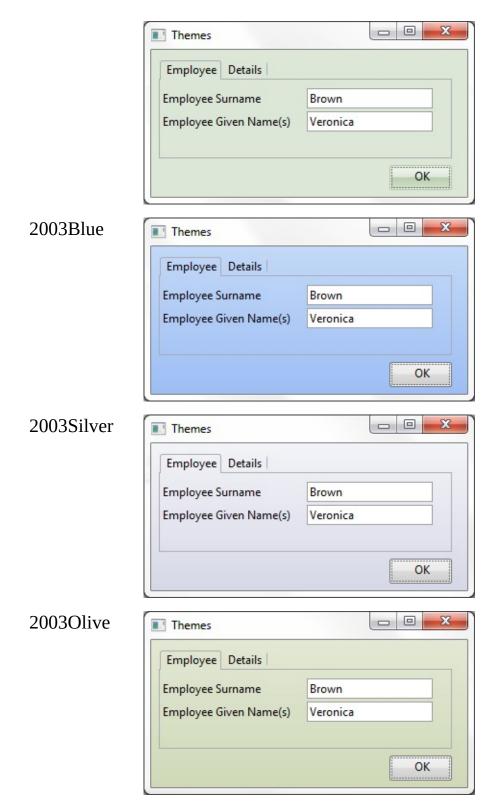
#### Theme Details/Example

Classic Default

Shell This theme queries the Operating System and chooses the best match (for example it will follow the Blue\Olive\Silver settings of a Windows XP PC).



2007Olive



2003Color

Themes	
Employee Details	
Employee Surname	Brown
Employee Given Name(s)	Veronica

You can customize the 2003Color themes. For more information see Customizable 2003Color Theme Type.

## **Theme Properties of Controls**

ThemeStyle Property

#### ThemeDrawStyle Property

Sometimes slightly different appearances may be required to get the most out of Visual LANSA themes.

The most obvious case is where group boxes and panels have been used as toolbars. Some themes have a dark control background which interferes with the display of icons (for example the 2007Graphite theme). In such cases use the ThemeDrawStyle to change the display of controls.

Image	The theming of container controls,(for example a tab folder as in this picture, which take up a large portion of the scree can be enhanced by specifying ThemeStyle Image.			
	Themes Employee Details Employee Surname Employee Given Name(s) Veronica OK ONly applicable to the 2007 themes.			
Toolbar	For controls used as toolbars, the Toolbar value will result in a better appearance.			
DarkTitle MediumTitle LightTitle	Used for labels that need to stand out in a form to appear correctly for the application theme. These values can also be used in places where visual style colors have been used to mimic theme behavior in Visual			
None	LANSA applications.			
Standard				

# **ThemeStyle Property**

The ThemeStyle property of controls allows a control to change its theme appearance.

Note that not all controls have this property.

None	The control appears without theme on a theme-enabled form.
Parent	Default. All controls follow the theme style of their parent and can only be themed if their parent form uses a theme.
Themed	Only has an effect if the form itself is theme-enabled.
Application	

# Customizable 2003Color Theme Type

You can use the #Sys\_Appln.ThemeType := 2003Color properties to customize the Office2003 theme colors.

#Sys_Appln.Theme2003Color1	Used as the base color for all backgrounds
#Sys_Appln.Theme2003Color2	Used as the base color for all highlighting
#Sys_Appln.Theme2003Colors	Can be used to initialize #Sys_Appln.Theme2003Color1 and #Sys_Appln.Theme2003Color2

You need to supply the base-color and highlight-color values:

Pumpkin
Rose
Eggplant
RainyDay
Spruce
Lilac

# Themes and VisualStyles

Themes and VisualStyles are closely related and on occasions can clash.

The theme settings do not override Visual Styles settings in an application. For example if a panel appears green because its VisualStyle specifies a NormBackColor of green, the panel will continue to appear green even in a 2007Blue themed application.

For cases where conflicts exist, you need to evaluate where the VisualStyle items should be reset in order to achieve the look you want.

## 6.30 Remember the User's Choices

When your application contains options and choices that your users are likely to want to change, make sure that they need to change them only once by recording them.

Similarly, often when users start up an application, they want to continue working with the same data they were working with when they last used the application. To make this easy the application should start up with the data that was being displayed when it was last shut down.

### **Storing Settings in the Windows Registry**

To store information about a form in the registry and then retrieve it, use the Change command with one of these special remembered values:

*remembered_value_for_user	Returns the value a field had when a particular user last executed this form
*remembered_value_for_user_in_system	Returns the value a field had when a particular user last executed any form or a function containing this field
<pre>*remembered_value_for_function</pre>	Returns the value a field had when this form was last executed
*remembered_value_for_system	Returns the value a field had when any form or function containing the field was last executed.

The first time you execute a Change command with one of these values, a registry entry for the last value of the field in question is created.

**Note:** These values are not available with a web interface as, for security purposes, they will not have access to registry settings.

#### Example

Our sample application is used to retrieve the name of an employee based on the employee number:

Employee Details	
Employee Number	A0070
Employee Surname	Brown
Employee Given Name(s)	Veronica

When the application is started, it shows the employee that was being displayed when the application was last closed down. The values are retrieved using this code:

```
EVTROUTINE handling(#com_owner.Initialize)
```

CHANGE FIELD(#EMPNO #GIVENAME #SURNAME) TO(\*remembered ENDROUTINE

#### **Remembering the Form's Size and Position**

You use this same method to remember a form's position and size, but first you need to create fields for storing the information:

```
Define #top *dec 7 0
Define #left *dec 7 0
Define #width *dec 7 0
Define #height *dec 7 0
```

Then you need to retrieve the last values of these fields and assign them to the appropriate properties of the form in the Initialize event:

```
EVTROUTINE handling(#com_owner.Initialize)
CHANGE FIELD(#top #left #width #height) TO(*remembered_value_for_fi
Set #com_owner top(#top) left(#left) width(#width) height(#height)
ENDROUTINE
```

To store the current values of these four properties to the fields when the application is closed, you need to specify:

EVTROUTINE handling(#com\_owner.closing) Change #top #com\_owner.top Change #left #com\_owner.left Change #width #com\_owner.width Change #height #com\_owner.height Endroutine

#### Simplifying the Code

You can simplify your code by grouping together all the fields you want the application to remember. You could call the group for example #Remember: group\_by name(#remember) fields(#top #left #width #height #empno #givenar

and then simply add this line to the Initialize event of the form:

Change #remember \*remembered\_value\_for\_user

#### **Making All Forms Remember**

You may want to consider adding the code required for remembering settings to the STD\_FORM component so that it will be in every new form you create. For more information refer to 6.10.4 Modify Default Component Behavior.

↑ 6. Creating Applications Using Components

## **6.31 Help Text and Documenting Components**

- 6.31.1 Writing Help Text
- 6.31.2 iHelpHandler Interface
- 6.31.3 Documenting Components

#### Also See

Repository Help Tab in the User Guide.

Using CHM files for Online Help Text example in the SET collection.

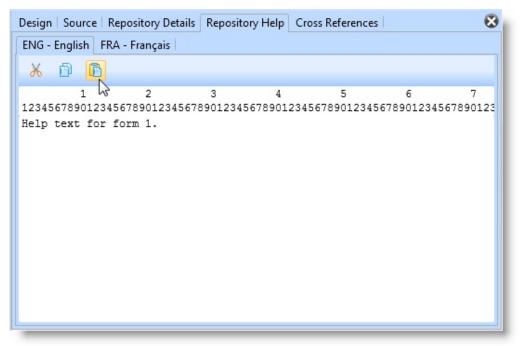
1 6. Creating Applications Using Components

# 6.31.1 Writing Help Text

You can write help text for fields, forms or reusable parts. In the case of forms and reusable parts you can also include in the help individual help topics for specific controls on the form or the part.

The help text will be displayed when the user presses F1 or clicks the Help button in the form's title bar during the execution of the application.

To write help text display the field/component list, the Repository Help tab:



Enter the help text, compile and execute the application. When the user presses F1, this is how the help is displayed:

Sample Application	×
$X \leftarrow \rightarrow$	
Contents       Index         Sample Application (Form)         Forms         Components         Processes         Functions         Fields	
Sample Application (Form)     Forms     Components     Processes     Functions	

In the case of a form or a reusable part, you can add help text for a particular control on the form or the part. To do this go to the last line of the help text for the language you are currently working with. Add a line that starts with \$\$COMPONENT= and then the name of the control to which the help is to be attached. For example:

```
$$COMPONENT=PHBN_1
```

Note that all text on this line has to be entered in capital letters and that it has to come before any \$\$LANGUAGE tags. Start the help text on the next line.

Depending on the circumstances, it is sometimes clearer to add all the help for all the individual controls in the body of the help so that all the topics can be found in the same place. In other contexts, especially when the help text is long, it is best to assign the help to a particular control.

You can also add images to the help text by entering the following keyword:

```
$$IMAGE=STOPLIGHT.BMP
```

If you add the image in the help text for the component, it must precede the \$\$COMPONENT keyword.

PHBN_1		
$\times \leftrightarrow \rightarrow$		
Index         Sample Application (Form)         HBN_1 (Component)         Forms         Components         Processes         Fields	Help text for button 1.	

## **Help Search Order**

This help is searched first for specific help for the individual controls, fields and reusable parts. If this does not exist the help for the form or reusable part is searched.

Note that in the case of controls that can contain other controls such as group boxes, tab sheets and panels, it is often a good idea to write the help text for the container control.

Thus, help can be defined for, say, a group box that describes how all the radio buttons operate within it. When help is requested for any of the radio buttons, the group box help is displayed instead.

↑ 6.31 Help Text and Documenting Components

# 6.31.2 iHelpHandler Interface

A flexible way of creating independent (i.e. not stored in the repository) help for components is to use the iHelpHandler interface.

Using the iHelpHandler interface you have the option of implementing whatever help mechanism you like. You simply need to plug in the appropriate piece of code and the help will be routed via a mechanism that sends notification that help was requested on a particular component. What you do after this is entirely up to you.

This example form implements the iHelpHandler Interface. When F1 is pressed, the focus component will determine whether it is implementing the help interface. If not, help will be handled through the normal automated repository help. If it is, the ProcessHelpRequest method will be executed.

Standard Number	0 \$	
Standard TEXT, SHORT		
Employee Number		
He	Standard_Text_Sh	hort DK

In the following code the Find\_Tag method receives a reference to the control that currently has focus. In the example, the ComponentTag property of the requesting component is used to determine the what help to show. If the component has a blank tag, the Handled map on the method is left as false, indicating to LANSA that help should go through the normal channels. However, if the component tag is not blank, it is displayed in a message box. Once control is returned from the message box to the form, the Handled map is set to True indicating that normal repository help should not be shown. Note that this example uses Full RDMLX:

Mthroutine Name(Find\_Tag) Help('Find a tag for the current control. Use Pare Define\_Map For(\*Input) Class(#prim\_objt) Name(#i\_Requestor) Pass(\*by\_ref Define\_Map For(\*Result) Class(#prim\_alph) Name(#o\_tag)

\* If a control is supplied If ((#i\_Requestor \*IsNot \*null) \*AndIf (#i\_requestor \*Is #prim\_ctrl))

\* If the Control has a tag, use it, otherwise go up the parent chain, and try again If (#i\_Requestor.ComponentTag <> ")

#o\_tag := #i\_Requestor.ComponentTag

Else

If (#i\_Requestor.Parent \*Is #prim\_ctrl)

\* Look for a tag in the parent
#o\_tag := #com\_owner.Find\_tag( #i\_Requestor.Parent )

Else

\* If not control supplied, use a default #o\_tag := 'DefaultTag'

Endif

Endif

Else

\* If not control supplied, use a default #o\_tag := 'DefaultTag'

Endif

Endroutine

When the ProcessHelpRequest method is being executed, you have complete

control over the application. Rather than using normal repository help if the tag is blank, it is quite possible to navigate the parent chain looking for a control that does have a tag. Thus with a simple recursive method it is possible to ensure that regardless of where focus is within a complex for a series of reusable parts, a help tag can be provided. The example Find\_Tag method demonstrates this technique.

If this approach to help is adopted within an application, it is necessary to ensure that is implemented for all forms and reusable parts. The recommendation is that the processing be implemented in a base class form and reusable part that all other components inherit.

iHelpHandler Code Example

↑ 6.31 Help Text and Documenting Components

### iHelpHandler Code Example

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM \*implements #Prim\_App.IHelp Define\_Com Class(#Std\_num.Visual) Name(#Std\_num) Componenttag('Stand Define\_Com Class(#Std\_texts.Visual) Name(#Std\_texts) Componenttag('Stanc Define\_Com Class(#Empno.Visual) Name(#Empno) Displayposition(3) Left(8

Mthroutine Name(ProcessHelpRequest) Options(\*Redefine)

\* Define\_Map For(\*input) Class(#prim\_objt) Name(#Requestor) Pass(\*by\_ref

\* Define\_Map For(\*input) Class(#prim\_boln) Name(#Handled)

#Handled := False

If (#Requestor \*IsNot \*null)

\* If the requesting component has a Tag, interrupt the help and use the user def If (#requestor.ComponentTag <> "")

\* Show Help for the requesting control #Com\_owner.Show\_help( #Requestor )

\* Set the Handled flag. If not set to true, normal Repository help will be displa #Handled := True

Endif

Endif

Endroutine

Mthroutine Name(Show\_Help) Define\_Map For(\*input) Class(#Prim\_Objt) Name(#Requestor) Pass(\*by\_refe

\* This example simply displays the Component tag property of the requesting ( \* However, the developer is now in full control of the help request, and can sho Use Builtin(Message\_Box\_Show) With\_Args(Ok OK Information 'Help' #requ Endroutine

End\_Com

1 6.31.2 iHelpHandler Interface

# **6.31.3 Documenting Components**

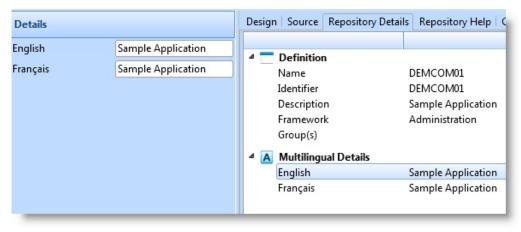
To ensure that your component can easily be reused, you should document it carefully. This is very important because without a detailed, up-to-date, accurate description it is unlikely the component will be used more than once.

A component is documented in its description and in the HELP parameter of RDMLX commands.

#### Description

When you create a component, you must enter a description for it in the New Component dialog. Obviously, a component's description should be as accurate and detailed as possible.

If you want to change the description later, you can do this in the Repository Details tab:



### **RDMLX Commands**

You can document developer-related information about your component using the HELP parameter of the BEGIN\_COM command.

BEGIN\_COM HELP('This form component contains a tabbed folder with tab sheets for employee contact information, salary and bank details and skills') LEFT(342) TOP(260)

This information is visible in the Help tab of the editor.

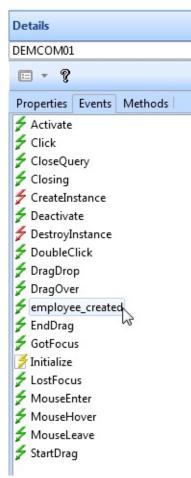
When you create custom-defined events, methods and properties, you document them using the HELP parameter of the DEFINE\_EVT, MTHROUTINE, DEFINE\_PTY and DEFINE\_MAP commands. These custom-defined events, methods and properties are made visible in the Details tab and in the Help tab of the editor.

For example, event with a mapped output value defined like this:

define\_evt name(employee\_created) help('This event is signaled whenever a new employy record is created. When this event is triggered, the employee number of the new employee is passed as THIS\_EMP value.')

define\_map \*output class(#empno) name(#this\_emp) help('help for value')

will show up like this in the Event tab of the Details tab:



The event and the value it passes are also visible in the Feature help:



## ↑ 6.31 Help Text and Documenting Components

# 6.32 GUI Guidelines

This section gives some general guidelines for developing GUI applications. You will find more detailed information about designing Windows user interfaces in the book Microsoft Windows User Experience (ISBN 0-7356-0566-1) by Microsoft Press and at the Microsoft user interface web site http://msdn.microsoft.com/ui.

6.32.1 Correct Use of Controls

6.32.2 A Few Rules for Navigation

6.32.3 Visual Design

6.32.4 Tab Sequence and Defaults

6.32.5 Hints

6.32.6 Loading Large Amounts of Data

6.32.7 Casual and Power Users

6.32.8 Plan Your Application

↑ 6. Creating Applications Using Components

### **6.32.1 Correct Use of Controls**

There are a number of conventions on the use of the different Windows interface controls. These conventions are based on the functionality of the control. For example you would quite naturally visualize an action or a command as a command button.

Often you will have a choice of several controls for visualizing a component. In deciding which is the most appropriate one look for the one which will do most to simplify the interface of your program. Also consider the other controls which have been placed on the form, so as not to have, for instance, more than one tree view in a window

Think about the Purpose of the Control

How Do I Represent a...?

There Are Only Three Ways for the User to Trigger an Action

↑ 6.32 GUI Guidelines

#### Think about the Purpose of the Control

Broadly speaking controls have the following purposes:

Action	Push-button Toolbar button Menu option	
Entry, display and selection	List view, list box and combo box	
	Grid	
	Tree view	
	Check box and radio button	
	Edit box, spin edit, track bar	
Feedback	Status bar and progress bar	
Grouping	Tab sheet, group box	
1 C DD 1 Comment II on a Community		

1 6.32.1 Correct Use of Controls

# How Do I Represent a... ?

The following guidelines are meant to help you choose the right visualization style for various situations. They are not absolute.

#### Choice between two options

Notice how differently data and logic are represented:

Database Data (Fields)	Program Logic (Actions)
To record a Yes/No choice in the data base, make the field's visualization style Check box.	To let the user choose from two actions, use
<ul> <li>Health fund member</li> <li>To record a choice of one of two values in the data base, make the field's visualization style Radio button.</li> <li>Male</li> <li>Female</li> </ul>	Add   Remove   Or use a menu.   Item   Add

#### Choice between several options

Again, notice how differently data and actions are shown:

Database Data (Fields)	Program Logic (Actions)
To record an user's choice of a value for a field, make the field's visualization style Picklist. The picklist's appearance can be: A set of radio buttons in a frame	To let the user choose from several items or actions, use a list with a set of buttons

Small Medium Large	Employee Give Employee Su aAbBcCd aAbBcCdDe Add
Employee Giv Employee Veronica Brown Fred Bloggs Fred Smithson Anne Simpson	Or use a menu for several items:
	Or for several actions:

# 1 6.32.1 Correct Use of Controls

### There Are Only Three Ways for the User to Trigger an Action

In GUI applications there are only three correct controls you can use to let the user trigger an action:

- A push-button
- A menu option
- A toolbar button

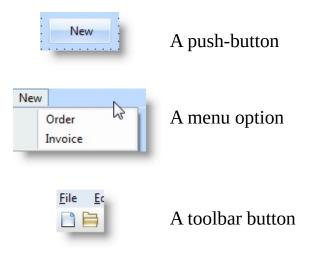
In standard GUI usage these three controls command immediate action. No other control has this characteristic.

Do not, for example, trigger an action when the user ticks a check box (except to write the choice to a database). This is because a check box is universally used to present a yes/no selection, not an action.

Let's say your application brings up a new window if the user decides to create a new invoice. It would be wrong to represent this as a check box, because the user does not expect a new window to appear when they tick a box:



For your application to behave predictably, you represent this choice as:



↑ 6.32.1 Correct Use of Controls

#### 6.32.2 A Few Rules for Navigation

- Consider whether navigation is really required. Does the functionality being implemented require a separate window?
- Supply a menu, command button or toolbar button for the end user to move to another window.
- Do not turn the menu structure of an existing LANSA process into a tree view. The purpose of a tree view is to display data.
- Do not use a check box or radio button for navigation. Their purpose of these controls is to record the choices an user makes.

### 6.32.3 Visual Design

Much of power of graphical user interfaces is based on the fact that visual patterns are quicker and easier to recognize than text.

Let's say you have a list of employees, some of them permanent, others are casual and you need to show them in one list. Most developers would create a list like this:

Employee	Туре	<b></b>
Arnold Wendy	Permanent	
Bergman Tom	Permanent	
Jacobs William	Casual	
Mason Anthony	Casual	
Milward Simon	Permanent	
Nguyen Stephen	Casual	
Norman Winston	Casual	
Ngo Jason	Permanent	
Olivera Anthony	Permanent	
Telavera Gordon	Permanent	
Watson Edward	Casual	-
•		•

There is nothing wrong with this list as such. However, to find out whether an employee is permanent or casual, the user needs to read the information in the Type column.

If the type of employment was identified visually, the user would be able to distinguish the permanent and casual employees at one glance.

Employee	Туре	<b></b>
Arnold Wendy	Permanent	
Bergman Tom	Permanent	
▷ Jacobs William	Casual	
Mason Anthony	Casual	
Milward Simon	Permanent	
▷ Nguyen Stephen	Casual	
▷ Norman Winston	Casual	
Ngo Jason	Permanent	
Olivera Anthony	Permanent	
Telavera Gordon	Permanent	
▷ Watson Edward	Casual	-
•		•

In this example it can be seen straight away that there are two different types of employees.

The icons used are arbitrary, but a frequent user will quickly learn to associate the icons with the type of employee.

If you wanted to, you could try to find more representative icons. However, it is

often hard or downright impossible to find icons that can communicate a meaning in an unambiguous manner, especially as most icons used in interfaces need to be very small.

## 6.32.4 Tab Sequence and Defaults

When you place components on a form, set the tab sequence of the components carefully, so that the user can move around the window using the Tab key.

In cases where the user is likely to choose one of the controls more often than the others, make it the default. For controls like radio buttons, check boxes and fields this is done using the tab sequence. For a push-button, this is done using the BtnDefault property.

#### 6.32.5 Hints

The hint (ToolTip) is a small pop-up window that provides descriptive text, such as a label, for a component. They are set using the Hint property of a component.

To make it easier for the user to learn your application, make sure that every control that needs explanation has a meaningful ToolTip. This is especially important if the control does not contain a caption, or if the caption is an icon.

### 6.32.6 Loading Large Amounts of Data

When you are choosing the visualization style for a component which will be handling large amounts of data, bear in mind that loading all of that data in one go may take quite some time. You may want to consider using controls which load only part of the data. For example if you create a component for a list of several thousand customers, you could adopt one of the following approaches:

- Use a standard reusable part as the basis The reusable part could contain an edit box where the user can type a partial name, a command button to fetch the entries that match that name and a list box that shows the entries.
- Show the customers grouped alphabetically in a tree view (in the same way as in the Visual LANSA Repository tab). When the user clicks on a letter, the customers whose name starts with that letter are fetched.

### 6.32.7 Casual and Power Users

Your choice controls will partly depend on the kind of user you are creating the application for.

If you are creating an application for casual users, you want to use controls which are easy to understand and which stop the them making wrong choices. Radio buttons and lists are good examples.

If you are creating an application that will be used repeatedly, choose the controls on the basis of how quick they are to manipulate. For example in many cases it may be faster to enter a value in field than choose it from a list.

In these applications use sound, animation and video very carefully because frequent users will grow tired of them.

### 6.32.8 Plan Your Application

A GUI interface offers great flexibility in structuring applications. Therefore you need to consider very carefully how you are going to build your application. The following steps are crucial in creating an interface that makes sense for the user.

- 1. Identify the applications business goals and the tasks the users will be performing. Think about the application's business goals and try to see if some of the user tasks can be eliminated or simplified.
- 2. Structure your application based on the user tasks. Group functions logically and, if possible, try to fit all the steps needed to accomplish one task in one window to minimize navigation overhead. However, avoid clutter.
- 3. Consider carefully how probable it is that your user will want to use individual options. If an option is commonly used, make it easily accessible, if it is seldom used, make it less visible.

# 6.33 Converting an Existing Application

This section shows how to convert an existing LANSA application using Visual LANSA. We use the LANSA Personnel demonstration system as an example.

6.33.1 Structure

- 6.33.2 Steps Required for Transforming an Existing Application
- 6.33.3 Converting a Work with... Type of Application
- 6.33.4 Showing Objects that Contain Lots of Details
- 1 6. Creating Applications Using Components

## 6.33.1 Structure

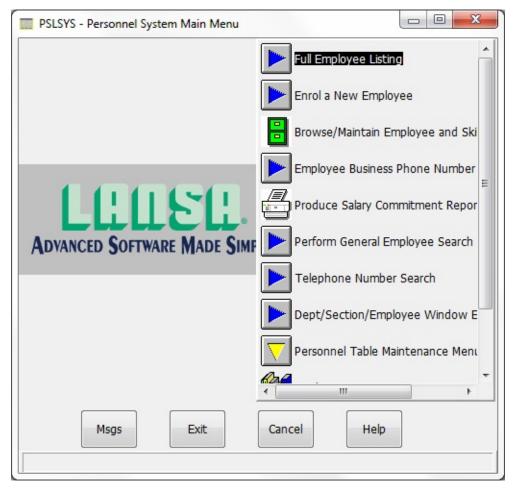
Visual LANSA applications consist of a set of windows (forms). There is no process with related functions as in traditional RDML programs instead you define the first window to appear in the application and then give the user access to the other windows in the application using menus or command buttons.

A well-designed Visual LANSA application will most likely contain fewer windows than a traditional LANSA application which means navigation between parts of the application is less of a concern. You will find that many applications consist of only one main window. This is because the controls, such as grids and tree views, take up little space on the screen while at the same time offering powerful functionality.

# Start-Up Windows

When you first start using Visual LANSA you might structure your application in the same way as a traditional LANSA application and provide access to various parts of the system in a menu similar to those used in iSeriiesgreen screen applications.

Below is the main menu of the of the LANSA Personnel Demonstration system modeled on the LANSA for the LANSA for i and Visual LANSA versions. It was created using Visual LANSA command buttons and text labels.



Even though this approach may be familiar to your users, it is not standard for Windows applications nor does it take advantage of the power of graphical user interface elements. It is also very inflexible: the user has to return to the menu every time he wants to perform a different task.

When complex systems are implemented in this way the result is usually several levels of menus with increased navigational overhead.

The other, more common solution is to provide access to functions using Visual LANSA drop-down menus. Menus are the de facto Windows standard for providing access to different functions.

t Edit Vie	w Tools Help
New	• Departme
Save	Section
Delete	Employee
Details	ion
Close	

The functionality of the menus is identical to the traditional menu, but there is one major difference: the menu bar contains several menus side by side. This means that you can fit up to about ten menus in one window. The menu bar only contains the menu titles and the user drops down the menu to display the menu options as necessary.

The menu bar in the example above consists of four drop-down menus: Employees, Skills, Departments, and Help. Each menu consists of one or more menu items. A menu item can also contain submenus (indicated by a triangular arrow).

Every menu title can be dropped down with the mouse or from the keyboard using the Alt key and the underlined letter in the title. Once the menu options are shown, the user can again either select them with the mouse or access them using the underlined letter and the Alt key. Commonly used options can also be given a direct shortcut, as in the above example the Find Employee function can be triggered using the Ctrl key and F without dropping down the menu. Creating consistent shortcuts is important because experienced users use the menus from the keyboard.

Notice that with menus much of the workspace of the window is left free because the menus take up very little screen real estate. Therefore it makes sense to add other elements to the window instead of using it simply as an entry point to the application. Typically you would add the most commonly used functions of the application to the start-up window to make it the main window of the application.

#### **Main Window**

An application in which all or most functions are accessed from the main window is easy and efficient to use because all functions are close at hand. (In a traditional IBM i menu driven application the user may have to navigate through several screens to accomplish just one task.)

<u>D</u> bject <u>E</u> dit <u>V</u> iew <u>T</u> ools	<u>H</u> elp		
dministration	- 196	IX %C	
All Objects	Employee	es of Administration	
Administration     Administration     Accounting     Administration     Administration     Purchasing     Fleet Administration     Group Accounts     Administration     Administration     Administration     Administration     Purchasing     Information Services     Legal     Momt Information	Number A1007 A1008 A1011	Surname Snell Sneddon Perron	Start Date 1/03/85 1/12/86 1/06/82

In the following Visual LANSA example the personnel system consists of a single main window which provides access to all the functions in the system.

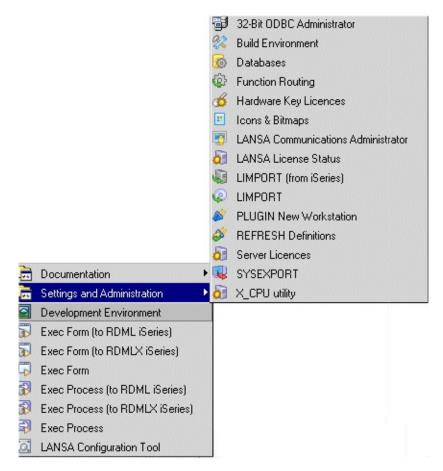
This implementation offers the user much more flexibility and easier access to functions than an application centered on a hierarchical menu.

The number of windows in the applications can be kept down because the components used to create the main window take up little screen space and have powerful functionality.

#### **Navigation Structure for Large Applications**

In cases where your application consists of several distinct functions, you should consider making them separate applications and then group them into one Windows program folder.

As an example, the LANSA program group in the Program menu contains one folder with a number of separate applications. These applications have their own icons and can be run independently or concurrently.



1 6.33 Converting an Existing Application

## 6.33.2 Steps Required for Transforming an Existing Application

The personnel system application shown in the example above is based on the LANSA personnel demonstration system with which you may be familiar. In creating it we had to decide how to transform the existing application into an efficient and easy-to-use GUI application.

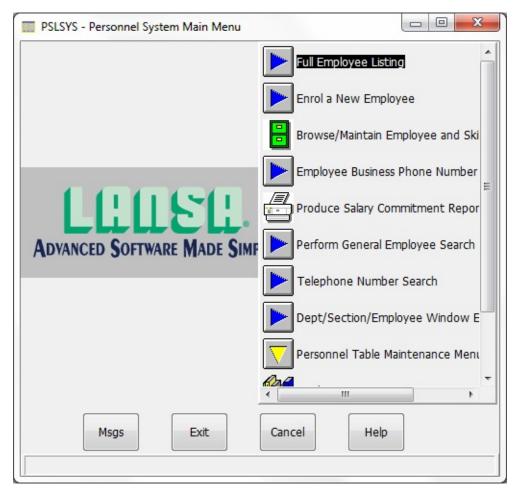
We had two choices: either redesign the entire interface or recreate the existing screens using the graphical elements. The latter choice would have resulted in a good looking application without any usability and user productivity benefits.

Our choice was to redesign the entire interface to make most use of the Visual LANSA capabilities. Much of the existing code could still be reused.

Creating this application involved three steps:

- Identifying the data and functions used in the application by examining the existing LANSA personnel system application.
- Choosing an appropriate visualization style for the components and creating them. All the components created for this system are reusable in other applications as well.
- Creating the application by putting the components on a form. This step took practically no time.

#### **Identifying Data**



The main menu of the LANSA personnel system provides access to various functions related with employees, skills, departments and sections. There are four kinds of functions: data entry, display, maintenance and printing.

In examining the system, we discovered the following main characteristics:

- Employees are organized in sections, and sections are grouped into departments
- The main purpose of the system is to record and display various details about the employees including salary and one or more skills
- The user of the system must be able to display, add, delete and change all the information

We came up with a list of about a dozen elements needed for the application. Because we thought it might be likely that these components could be reused in other applications, we decided to make them reusable parts.

### **Creating the Components**

We chose to build the interface around a list of employees. The employees

shown would depend on the department and section selected. It would also have to be possible to find an employee by name.

# List of Employees

The list of employees consists of employee names and employee numbers. We had the following choices for its visualization style:

- A list box presents lists of items from which the user can make a selection. The text in the list box cannot be changed which made it unsuitable for our purposes.
- A combo box consists of an edit box combined with a list box. New text can be entered in the edit box and when an entry in the list is selected, it is shown in the edit box and it can be edited.
- A tree view shows a hierarchical view of data. We could have created a tree view showing employees with their skills. However, because the tree view was even better suited for displaying departments and their sections, we reserved it for this use. A window with two tree views could become confusing.
- A grid shows columns of data which can all be edited. We did not want to let our users to change the employee number which made the grid unsuitable.
- The list view shows a list of items in one or more columns. The text in the first column from the left can be edited. The list can be sorted by any column. The items can be preceded by a small icon.

We chose the list view because it allowed the direct editing of the employee name but not of the employee number, and because the user could sort it either by employee name or employee number.

# **Departments and Sections**

The obvious visualization style for departments and sections was a tree view because the relationship between departments and sections is hierarchical (departments consist of several sections).

The nodes of the tree can be expanded and collapsed by clicking on them. The nodes can be preceded by icons as in our example.

# Skills

This list of skills had to be visualized as a grid, because we wanted to give the user the ability to change all the skill-related information (skill, grade, qualification date).

# Salary

The salary was visualized as a edit box with spin buttons. The user can change the salary using the spin buttons.

# Health Fund Member and Employment Type

The health fund member field was visualized as a check box because it represents a Yes/No choice.

The employment type (part-time and full-time) was visualized as radio buttons. Radio buttons are normally used to set a mutually exclusive choice. Therefore it should always be used in groups of two or more.

# Post/Zip Code

We chose to make the post code field a combo box because in this example we assume that there is only a limited set of post codes in use (in real life this would probably not be so). We chose a combo box instead of a list box because a combo box can be defined as 'drop-down'. This means that the list part of the combo box is hidden until the user clicks on it.

# Menu

We chose the MINSTDBTN menu bar as the basis for our menu. It contains the File, Edit, and Help menus.

We modified the New option of the File menu to allow the user to choose between new employee, section, and department.

# **Creating the Application**

Once we had created all the components, creating the application was only matter of creating a form to be the main window of the application and placing the components on it.

↑ 6.33 Converting an Existing Application

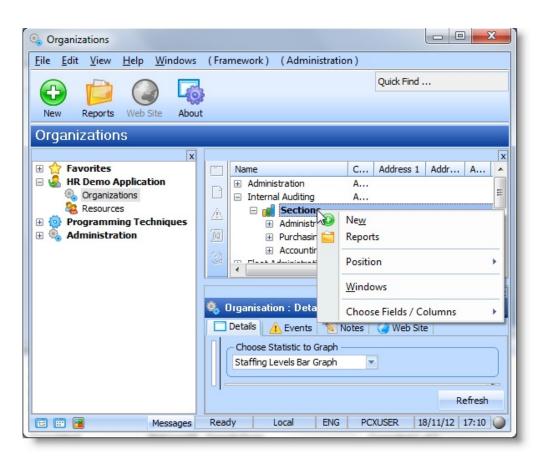
# 6.33.3 Converting a Work with... Type of Application

In the iSeriiesworld the Work with... type of applications are very common. In them you are presented with a list of items and several options (represented by numbers) for actions that can be performed on any of the list items.

		4=Delete	Spooled Files 5=Display ing status		elease	7=Messa	ges
Opt File QSYSPRT QPJOBLOG QSYSPRT QPJOBLOG QPJOBLOG QPJOBLOG QPJOBLOG QPJOBLOG QPJOBLOG QPJOBLOG QPJOBLOG QPJOBLOG	User Qothprdown Qothprdown Qothprdown Qothprdown Qothprdown Qothprdown Qothprdown Qothprdown Qothprdown Qothprdown	Device or Queue PERMANENT PERMANENT PERMANENT PERMANENT QEZJOBLOG QEZJOBLOG QEZJOBLOG QEZJOBLOG QEZJOBLOG	User Data r75_list01 r75_list01 r75_list02 r75_list02 SERVERWK SERVERWK SERVERWK SERVERWK SERVERWK SERVERWK	Sts RDY RDY RDY RDY RDY RDY RDY RDY RDY RDY	Total Pages 2 1 2 1 1 1 1 1	Cur Page	Copy 1 1 1 1 1 1 1 1 1 1
Parameters for ===> F3=Exit F10='	options 1, 2, View 3 F11=V			=Prin	ters F	24=More	
M <u>A</u> a							09/003

### A List and Menus

In a GUI environment, this basic paradigm is typically presented as a list view and menu options. The menu options apply to the currently selected option and when selected they bring up additional windows..



The basic approach is the same as with the Work with... interface, only the way of invoking the actions is different.

#### **Explorer-like Interface**

This kind of interface can easily be extended to handle more complex views of data by adding a tree view to the left. The tree view is used to navigate through hierarchies of data.

Again, the actions are performed using menu bars or pop-up menus. Another example of this kind of interface is the Windows Explorer.

The GUI list interface has several advantages over the Work with... interface:

- You can precede the list items with icons to make it easier to distinguish between different objects
- In a list view, you can offer the end user four views of the list items:
  - big icons
  - small icons
  - list
  - report

- You can represent hierarchical relationships visually
- The user does not need to match an action to a number
- The user can sort information by any column in the list simply by clicking on the column heading.

↑ 6.33 Converting an Existing Application

# 6.33.4 Showing Objects that Contain Lots of Details

If your application is dealing with objects that have lots of details, use tabbed folders.

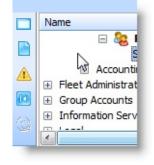
			Status	
SOA concepts	Learn how to use SOA entry points to create an	15/06/2007	Not Attending	
Tivoli Nation J	Join your peers to learn, share and build a new I	30/06/2007	Under Consideration	1
Linux tools E	Extend markets, master your most urgent busin	14/06/2007	Not Attending	=
Boffinfest J	Join top technical experts and learn about the la	23/06/2007	Registered for	1
Websphere /	Attend this premium event specifically designed	27/07/2007	Under Consideration	
			Unspecified	
			Unspecified	1
			Unspecified	-

Tabs provide a means to logically group large amounts of information and they provide easy navigation.

Be careful where you place command buttons on a tabbed folder: if a button applies to all the tabs, place it outside the tabbed folder so that the button's scope is visually clear. If the button only applies to an individual tab, put it on the tab itself.

### Toolbars

You can represent the most common actions for the list items as a toolbar.



 $\ensuremath{\Uparrow}$  6.33 Converting an Existing Application

# 7. ActiveX Controls

ActiveX controls are third-party standard reusable components, which you can use in any application that supports ActiveX technology. Visual LANSA provides full support of ActiveX.

There are ActiveX controls for instance for calendars, text formatting, graphing, spell checking, advanced grids or web browsers. You can also integrate many applications, for example the Microsoft Office Suite (Word, Excel, PowerPoint etc.) into your Visual LANSA application using ActiveX technology. (For more information, see 7.2 Integrating with ActiveX-Enabled Applications.)

LANSA provides support for the use of ActiveX controls and COM automation and the option of wrapping LANSA components as ActiveX components.

7.1 Using ActiveX Controls

7.2 Integrating with ActiveX-Enabled Applications

7.3 ActiveX Wrapping

# 7.1 Using ActiveX Controls

To use an ActiveX control, you need to first install it on your computer. After this you need to enroll it in the LANSA repository as an ActiveX component.

You work with ActiveX controls the same way as you work with any other controls: by using the control's properties, methods and events. Some controls are very simple and little programming is required to use them, others provide complex functionality and may require much more coding than is needed than when using native Visual LANSA controls.

Active-X Exception Handling traps failed HRESULT values at runtime so that they do not cause application failures. Access is provided to the HRESULT values. For more information, refer to COM Error in the *Visual LANSA Feature Help*.

- 7.1.1 Before You Decide to Use an ActiveX Control
- 7.1.2 Obtaining ActiveX Controls
- 7.1.3 Enrolling an ActiveX Control in the Repository
- 7.1.4 Pre-Enrolled ActiveX Components
- 7.1.5 Viewing the ActiveX Control
- 7.1.6 ActiveX Terminology
- 7.1.7 Using an ActiveX Control
- 7.1.9 Licensing and Deployment
- ↑ 7. ActiveX Controls

### 7.1.1 Before You Decide to Use an ActiveX Control

When you decide to use an ActiveX control within your LANSA application you need to know that:

- You are introducing a dependency into your application over which the LANSA product has no control. Such a dependency may not be obvious or may not appear until sometime in the future.
- LANSA support is not obliged to help you resolve any problem with ActiveX controls. They may or may not have information available to assist you.
- Any problem you report that involves an ActiveX control must be reproducible on your development Visual LANSA system. Problems related to ActiveX controls deployed to non-development systems should not be reported.
- ↑ 7.1 Using ActiveX Controls

# 7.1.2 Obtaining ActiveX Controls

Most likely you have on your PC some ActiveX controls, which are part of your operating system or the software you have installed. For example, if you have installed Microsoft Internet Explorer, you will have the Web Browser ActiveX control. You can embed this control in your form to integrate IE web-browsing functionality in your application.

Most ActiveX controls are purchased separately from a variety of vendors. Many of them can be obtained from Web sites and you can usually download them for evaluation. Some sites where you can obtain them are:

www.activex.com

www.activexcontrol.com

ActiveX controls need to be installed on the PC, and usually the controls come with installation instructions. During the installation, information about the control is recorded in the Windows registry.

↑ 7.1 Using ActiveX Controls

# **Downloading and Installing an ActiveX Control**

To show you the basics of how to install an ActiveX control in your LANSA system, we use a keyboard control created by Iocomp Software.

The keyboard control is part of a collection of instrumentation ActiveX controls. It is an on-screen keyboard with mouse control support and is designed for touch-screen applications or keyboard-less applications:

KEYBCTRL - Keyboard control	×
Esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F1	12 psc slk brk
1 2 3 4 5 6 7 8 9 0 - = bksp	o ins hm pup nlk / * -
Tab q w e r t y u i o p [ ] \	del end pdn 7 8 9
Lock a s d f g h j k l ; ' Enter	ar 4 5 6 <sup>+</sup>
Shift z x c v b n m , . / Shift	U 1 2 3
Ctrl W Alt Alt U L Ctr	ILDRO.

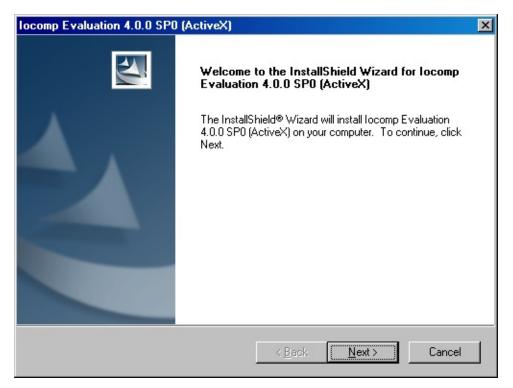
You can download an evaluation copy of the controls from the iocomp software site.

Iocomp Software - In:	strumentation,	Scientific <u>,</u> ar	d Engineering Co	mponents <u>- M</u>	icrosoft Internet	Explorer
<u>F</u> ile <u>E</u> dit <u>V</u> iew F <u>a</u> vor	rites <u>T</u> ools <u>H</u>	<u>l</u> elp				
⇐ Back 👻 🔿 👻 🛃	ର 👌 🖓 📿 Sea	arch 🛛 🙀 Favo	rites 🛞 Media 🔮	) B- 31	ei 🗉 🎎	
Address 🕘 http://www.ioo	comp.com/Downl	oads/Evaluation	ns.aspx			• @
Google G-		- ] Go o 🦟 🧖	🍟 👻 👷 Bookma	arks <del>v</del> 🔊 150	blocked 🏻 券 Che	eck 🗸 »
0000						
	RE					
iOCOI	nr					
	Home	Products	Downloads	Support	Corporate	Orde
	<u>Home</u> »	<u>Downloads</u> >	» Evaluations			
Evaluations						
Examples	Evalua	tions				
Updates						
CAB Files	ActiveX	VCL	.Net WinForms	.Net Comp	act Framework	
Visual C++ Wrappe	er <mark>s</mark>					
Betas	-	_				
Documentation						
Misc	V4 SP0					
Translations	All Develo	oment Envir	ronments : <u>Dowr</u>	nload		
Demos		1		Verane.		
ActiveX/VCL	Additional	l Downloads	.,		•	
.Net WF	Additional	i Dowinioaus	••			
.Net CF	Gettin	<u>a Started Ma</u>	anual (Right-Click	to Save)		
	Gettin	<u>g Started Ex</u>	amples			
	23 23					3
Quick Links						
>> Evaluations			- manana m	and the second		

To download the ActiveX controls, select the ActiveX tab and click on the Download link.

When the Security Warning dialog is displayed, click on the Save button. In the Save As dialog specify where the control is to be saved, for example in the Download directory.

The instrumentation controls are downloaded in an exe file. Execute the file by double-clicking it. Install the controls:



Accept the terms and conditions, register your details and choose the platform:

locomp Evaluation 4.0.0 SI	PO (ActiveX)	×				
Registration (Part 2 of 2) Technical Information         Select the Operating Systems and Software Development Environments         that you plan on using or wish to use with our component products						
OS Windows  95 NT 4.0  98 2000  ME V×P  2003 Vista 2008	Development Environments         VB6       Delphi 4         Visual C++ 6       Delphi 5         Microsoft IE       Delphi 6         MS Word       Delphi 7         MS Excel       Delphi 2005         VS2003       Delphi 2005         VS2005       Turbo Delphi         VS2008       Delphi 2007         VLANSA       Delphi 2007	<ul> <li>C++ Builder 4</li> <li>C++ Builder 5</li> <li>C++ Builder 6</li> <li>Turbo C++</li> <li>C++ 2007</li> <li>BDS2006</li> <li>RAD Studio 2007</li> </ul>				
Type In Your Comments Here:	< <u>B</u> ack <u>S</u> end >	Cancel				

Accept the default settings in the dialogs that follow.

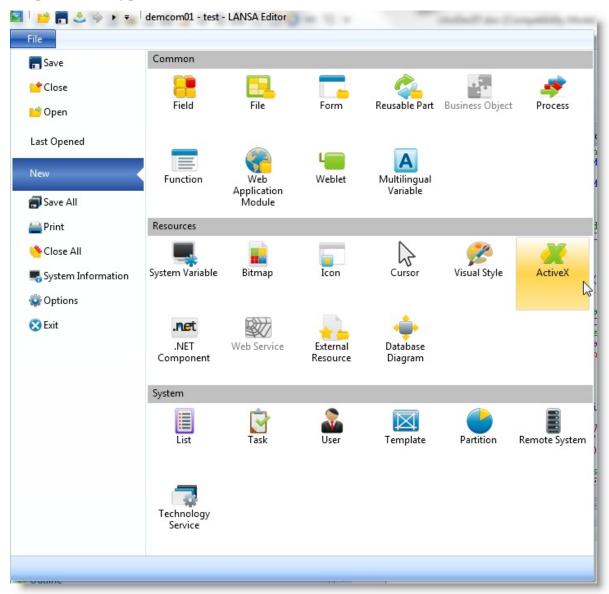
After the installation is complete, the ActiveX controls are installed in the Windows registry of your computer. They are now ready to be enrolled in the

LANSA Repository. ↑ 7.1 Using ActiveX Controls

# 7.1.3 Enrolling an ActiveX Control in the Repository

After an ActiveX control has been installed on your PC, you need to enrol it in the Repository as a component of type ActiveX before you can use it.

To enrol the keyboard control downloaded in the previous section, create a new component of type ActiveX.



Call it for example KeybCtrl. Always store ActiveX controls in the group ActiveX Controls so that they will be easy to locate in the editor.

🦉 New ActiveX		×
Name Description Framework Group Identifier	Administration (ADMINISTRATION)	<u>Create</u> Ca <u>n</u> cel

When you have created the component, display its Properties tab in the editor. Make sure the Ancestor property of the component is #PRIM\_MCVO:

				<u> </u>
KEYBCTRL				-
Properties	Events N	lethods		
Ancestor			#PRIM_MCVC	

Then use the component's ProgID property to specify the ActiveX control to be enrolled in this component.

Click on the button with three dots. The ActiveX Component Browser window is displayed.

The browser shows ActiveX Controls (visual components) and Other Components (non-visual). The other components include ActiveX-enabled applications which can be driven through properties and methods they expose.

Click on Controls to display all the available ActiveX controls on your PC.

ActiveX Component Browser	×	
⊕ adbanner Class		
Adobe Acrobat Control for ActiveX		
BDE50.ctrlBDE50		
⊞ Billboard Class		
⊕ Calendar Control 9.0		
CDDBAOLControl Class		
⊕ CerberusCDPlayer Class		
⊕- CheckBox Class		
COMNSView Class		
⊡ Crystal Report Viewer Control		
🕀 CrystalAnalysis Class		
🕀 CrystalGridControl Class		
D/IQU//oldered Object	-	
<u>D</u> K <u>C</u> ancel		

The name of the keyboard component is iKeyBoardX Control:

ActiveX Component Browser	×	
i iAnalogDutputX Control		
庄 - iAngularGaugeX Control		
庄 - iAngularLogGaugeX Control		
⊞ iCompassX Control		
⊡ iDualCompassX Control		
iGradientX Control		
⊡ ilmageDisplayX Control		
iKeyBoardX Control		
⊞⊸ iLabe⊠ Control		
⊡ iLCDMatrixX Control		
iLedDiamondX Control		
⊡ iLedMatrixX Control		
il ad Rostanalo Y Control		
<u>QK</u> <u>Cancel</u>		

Select it and click OK.

The ProgID and the TypeLibId properties of the control are filled in. Other

properties for the size, position etc. of the control become visible. The control itself is shown in the Design tab.

KEYBCTRL - Keyboard Control		×
Esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10F11F12	psc slk brk	
1 2 3 4 5 6 7 8 9 0 - = bksp	ins hm pup	nlk / * -
Tab q w e r t y u i o p [ ] \	del endpdn	7 8 9
Lock a s d f g h i k l ; ' Enter		4 5 6 +
Shift z x c v b n m , . / Shift	U	1 2 3 nte
Ctrl W Alt Alt U L Ctrl	LDR	0.

Save the ActiveX control. It is now enrolled in the repository.

After you have enrolled an ActiveX control, you can inspect its properties, events and methods. You can also change the values of the properties of the enrolled control (both LANSA properties and its built-in properties) to set default values for it. These values are used as defaults when you use the ActiveX component in your LANSA form or reusable part.

The control is now ready to be used in your LANSA applications. After you have enrolled an ActiveX control in a component, you would typically only reopen the component if you wish to change any of its default properties.

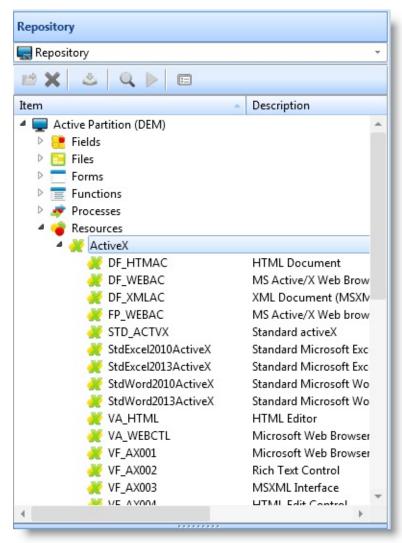
↑ 7.1 Using ActiveX Controls

## 7.1.4 Pre-Enrolled ActiveX Components

Visual LANSA is shipped with some pre-enrolled ActiveX components. You can find the enrolled controls in the group ActiveX Controls in the Repository tab in the Resources node.

The name of the pre-enrolled components starts with VA\_.

These components are only usable if you have the required software installed on your PC. In other words you can use, for instance, the VA\_WORD component which enrolls Microsoft Word ActiveX component in the LANSA repository only if you have installed Word on your PC.



#### ↑ 7.1 Using ActiveX Controls

## 7.1.5 Viewing the ActiveX Control

When you have saved an ActiveX component, all the events and methods of the enrolled ActiveX control are visible in the Details tab. Here are the methods for the keyboard control downloaded in the previous sections.

EYBCTRL		<u> </u>
		<u> </u>
Properties Events Methods	N2	
AboutBox	OPCRemoveAllItems	
BeginUpdate	Realize	
EndUpdate	RepaintAll	
CetBytesJPEG	SaveImageToBitmap	
GetBytesPNG	SaveImageToJPEG	
CetObjectPointer	SaveImageToMetaFile	
GetOPCItemActive	SaveImageToPNG	
GetOPCItemAutoConnect	SavePropertiesToFile	
GetOPCItemComputerName	SetFocus	
GetOPCItemData	SetFocusHandleToActiveWindow	
CetOPCItemGroupName	RetOPCItemAutoConnect	
CetOPCItemItemName	SetOPCItemComputerName	
CetOPCItemPropertyName	🔀 SetOPCItemData	
CetOPCI temQuality	SetOPCItemGroupName	
CetOPCItemServerName	SetOPCItemItemName	
CetOPCItemTimeStamp	SetOPCItemPropertyName	
CetOPCItemUpdateRate	SetOPCItemServerName	
reaintToDC	🖄 SetOPCItemUpdateRate	
🖉 LoadPropertiesFromFile	🖄 ShowMessages	
Lock	🖄 Unlock	
OPCAddItem	🔀 Unrealize	
OPCDeleteItem	🔀 UpdateDisplay	
OPCItemActivate		
OPCI tem Activate Suspend		
OPCItemCount		
OPCI tem Deactivate		
OPCItemLoaded		
OPCItemUpdateResume		
OPCItemUpdateSuspend		

The events and methods shown are defined by the control itself except for a few general ones which are common to all controls.

The Properties tab for the ActiveX component shows its Visual LANSA properties and its built-in properties. The LANSA properties determine for instance the size and the default position of the control.

To see the built-in properties of the control, use either the the Properties

## Interface of the control itself or the Component Features (F2).

		₽ × 🦽
KEYBCTRL		▼ 2
Properties Events Methods		<b>▼</b> Outline
Ancestor	#PRIM_MCVO	
AutoFrameRate	False	
BackGroundColor	0	Details
BorderStyle	KEYBCTRL.ibsNone	0
BusyUpdates	Wait	
BusyUpdatesOfParent	False	2
ComponentClassName	KEYBCTRL	Repositor
ComponentDescription	Keyboard Control	Site
ComponentFramework	ADMINISTRATION	ALIO ALI
ComponentGroup	ACTIVEX CONTROLS	
ComponentHandle	1902696	
ComponentPatternName	KEYBCTRL	Favorites
ComponentTag		Orii
ComponentTypeName	KEYBCTRL	S.
DisplayPosition	1	
DragStyle	None	
Enabled	True	
Enabled_COM	True	
FocusHandle	0	
FontColor	0	
Height	167	
Height_COM	167	
Hint		
HintShow	True	
HintShowOfParent	True	
Left	0	
Name	KEYBCTRL	
OptionSaveAllPropert	False	
OuterMargin	5	
Owner	#KEYBCTRL	
PopupMenu	*NULL	

## ↑ 7.1 Using ActiveX Controls

## **Properties Interface**

Many ActiveX controls come with a user-interface for manipulating their builtin properties. To see whether a control has such an interface, select the component and right-click to open its pop-up menu. Check if it contains a Properties option. (In some controls this option is called Edit or a similar name).

KEYBCTRL - Keyboard Control		×
Esc F1 F2 F3 F4 F5 F6 F7 F	8 F9 F10 F11 F12 psc slk brk	
<u>123456789</u>	0 - = bksp ins hm pup	nlk / * -
Tabqwertyui	op[]\ delendpdn	789
Lock a s d f g h j k	I ; ' Enter	4 5 6 +
Shift z x c y b n m .	, . / Shift U	123
Ctrl W Alt	🖌 🎨 <u>F</u> eatures : KEYBCTRL F2	0
	What's This?	
	Add Popup Menu	
	Goto Definition	
	Save Definition	
	Save PRIM_MCVO Definition	
	Properties	
	▶ Events : KEYBCTRL	

The Keyboard Control has a Properties interface:

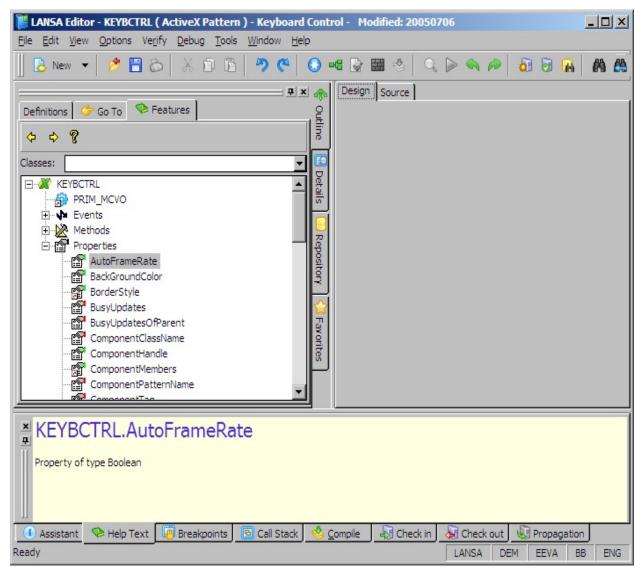
iKeyBoardX Control Properties	×
Control About	
Control Keys Theme	
Style 104 Keys  Second Row Margin 10  Properties  Load  Note: Changes will be applied immediately to the component and	
Outer Margin 5	
Font © None © Raised © Lowered	
Update Frame Rate 60 🚽 🗖 Auto Frame Rate 🛛 BackGround Color 🦳	
OK Cancel Apply	

You can use this interface to set the properties of the Keyboard control. The property values you set for an ActiveX control when you enrol it will be the default values when the control is included in a form or a reusable part.

↑ 7.1.5 Viewing the ActiveX Control

## **Component Features**

As with native Visual LANSA controls, you can view a description of an ActiveX control's properties, methods and events by selecting it and displaying the Features tab.



The help provided for ActiveX controls through the Features help is brief (it is provided as part of the ActiveX control itself). It gives you an indication of the purpose and type of the properties, methods and events of the control.

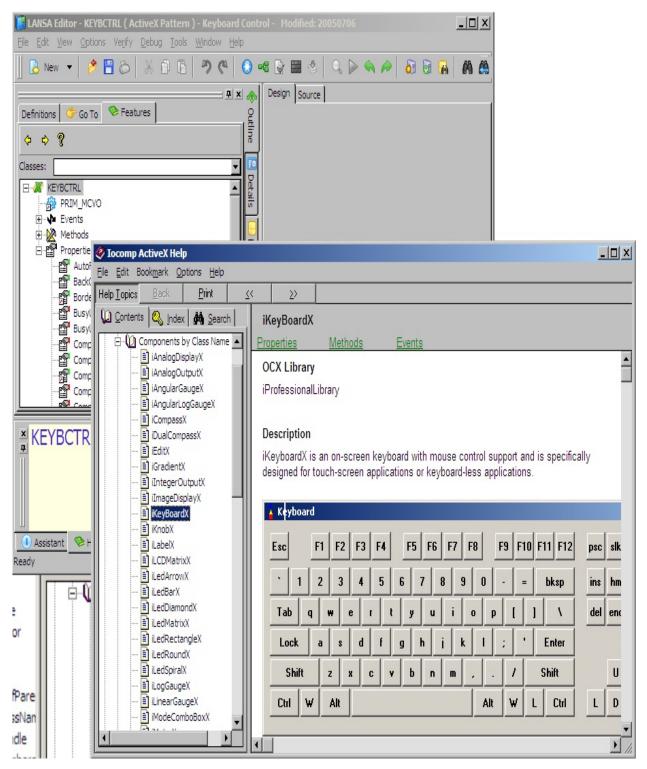
The Features shows you all the properties (both Visual LANSA and built-in), events and methods of the component. The Visual LANSA properties are used to set attributes which are common to all controls such as their height and the width. The built-in properties are specific to the ActiveX control.

When viewing ActiveX controls, you will come across properties which end with \_COM. This is because the \_COM suffix is added to the control's native property if there are Visual LANSA and built-in properties which are synonymous. For example the keyboard control has both a LANSA and a builtin property Width (Width\_COM). Often properties that have the same name do similar things. Read the help for both properties to decide which one to use.

 $\Uparrow$  7.1.5 Viewing the ActiveX Control

## **Component Help**

Many (but not all) ActiveX controls also come with their own extended help. To see the help, select the control in the Feature Tab and press F1.



If a control has context-sensitive help, you can select one of its properties, events or methods in the Features tab and press F1 or click on the help button on the toolbar of the Features window to display help for it. If the help is not context-sensitive, you must first select the control itself in the Features window and then display the help.↑ 7.1.5 Viewing the ActiveX Control

## 7.1.6 ActiveX Terminology

You do not necessarily need to know much about ActiveX terminology in order to use an ActiveX control in your application, but here is some basic information you may be interested in.

When you enrol an ActiveX component in the Repository, you are defining two things: the Type Library that contains the control and the control itself defined by its ProgID.

↑ 7.1 Using ActiveX Controls

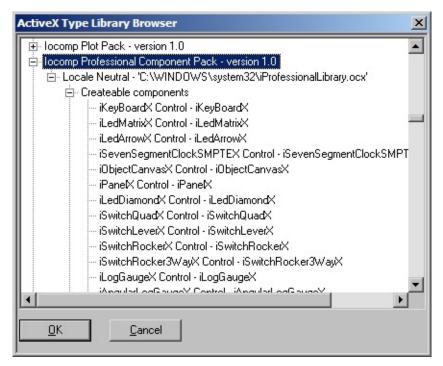
## **Type Library**

When you enrol an ActiveX component by specifying its **ProgID** property, in most cases LANSA automatically determines the type library which contains this component. This means that you do not normally need to worry about specifying the type library of a component.

A type library can contain more than one component. For example the type library that contains the keyboard control also contains other controls. You can see this if you select the Keyboard control's TypeLibId property in the Properties tab of the component and click on the button with three dots. Its type library is displayed.

ActiveX Type Library Browser	×
⊡ locomp OPC Common Library - version 1.0	•
⊡ locomp Plot Pack - version 1.0	
Iccomp Professional Component Pack - version 1.0	
⊞ IP NAT Helper - version 1.0	
⊞- IS DA0 351 Object Wizard - version 1.0	
🗄 IS DAO 36 Object Wizard - version 1.0	
i ISAFrdm 1.0 Type Library - version 1.0	
⊞- ISDownloadManager 1.0 Type Library - version 1.0	
⊞- ISDSStorage - version 1.5	
⊞- ISDSStorageOld - version 1.5	
⊞- ISIDEProject Type Library 1.0 - version 1.0	
⊞- ISLabel - version 1.0	
⊞- ISList - version 1.0	
⊞- ISLookup - version 1.0	
⊞- ISMoniker 1.0 Type Library - version 1.0	
⊕ ISMSDE2KWizard 2.0 Type Library - version 2.0	-1
in JakkaiEntiku 1.0 Turoo Libroru. uoraione 1.0	
<u> </u>	

As you can see, the name of the type library of the keyboard control used in the previous examples is Iocomp Professional Component Pack. Expand the items under the library. Notice it contains several creatable components.



You will be able to access the other controls in the type library using this component.

Internally the purpose of type libraries is to provide an interface which lets applications know how to interact with the ActiveX component. The library describes in detail the classes in the component and their methods, properties, events, constants etc.

To see the contents of the Iocomp Professional Component Pack type library, select the keyboard control and then display its Features help (F2). In the Features window drop down the list of classes in this library:

😿 K <mark>F</mark> ont	Interface
FONTBOLD	Alias
FontEvents	Interface
FONTITALIC	Alias
FONTNAME	Alias
FONTSIZE	Alias
FONTSTRIKETHROUGH	
	Alias
	CoClass
iCompassX	CoClass Interface
IDispatch	CoClass
iDualCompassX IEnumVARIANT	Interface
IFont	Interface
IFontDisp	Alias
IFontEventsDisp	Alias
IiAngularLogGaugeX	Interface
IiAngularLogGaugeXEvents	
liCompassX	Interface
<b>IiCompassXEvents</b>	Interface

A type library contains these types:

- CoClasses Defines classes that can be used to create objects. A CoClass has events, methods and properties.
- Interfaces Defines a set of methods and properties. A coclass implements an interface to use the interface's methods and properties.

Enumerations Constants used to make code more readable.

- Aliases Alternative names used to make code more readable.
- ↑ 7.1.6 ActiveX Terminology

## ProgID

The ProgID (programmatic ID) parameter is the human readable name of an ActiveX component, for example iProfessionalLibrary.iKeyboard.

The ProgID is an alias of the class ID of an ActiveX component. The class ID is a globally unique identifier (GUI), a 128 bit value guaranteed to be unique, represented as a 16-digit hex number.

KEYBCTRL	: <u>4</u> ]	
Properties Events Met	nods	Outline
Enabled_COM	True	
FocusHandle	0	
FontColor	0	Details
Height	167	S
Height_COM	167	
Hint		
HintShow	True	l e
HintShowOfParent	True	Repository
Left	0	12
Name	KEYBCTRL	
OptionSaveAllPropert	False	2
OuterMargin	5	
Owner	#KEYBCTRL	Favorites
PopupMenu	*NULL	3
ProgId	iProfessionalLibrary.iKeyBoard	
ScreenLeft	481	
ScreenTop	211	
SecondRowMargin	10	
Style	KEYBCTRL.ikbs104	
TabPosition	1	
TabStop	True	
Тор	0	
TypeLibId	{A8B345A0-74B5-11D3-85C2-001	
UpdateFrameRate	60	
Visible	True	
Visible_COM	True	
VisualStyle	*NULL	
VisualStyleOfParent	True	
Width	540	
Width_COM	540	

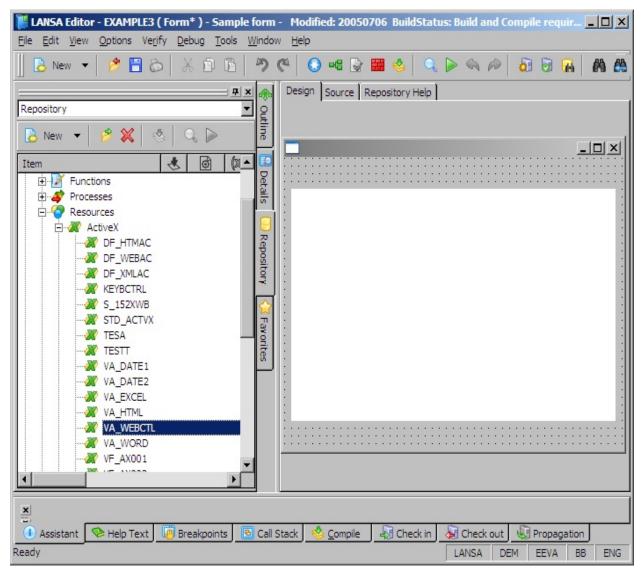
#### ↑ 7.1.6 ActiveX Terminology

## 7.1.7 Using an ActiveX Control

You use ActiveX controls in your application in the same way as you use native Visual LANSA controls.

This example shows how to use the Microsoft Web Browser ActiveX control which is installed on your PC if you have Microsoft Internet Explorer.

To try out the Microsoft web browser control, create a form. In the editor open the ActiveX Controls in Resources node of the Repository tab. Locate, select and then drag and drop the VA\_WEBCTL control on your form (as always, you need to have the Design tab topmost to drag and drop a control).



Now you have a form with a ready-to-use web browser on it.

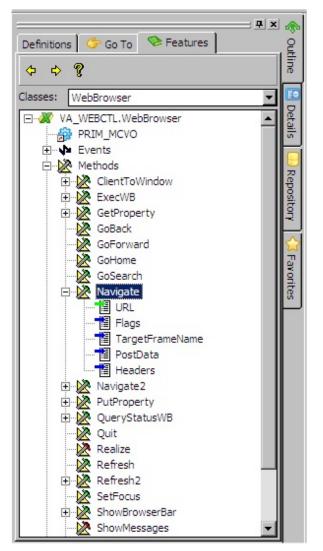
Display the Source tab of your form. As you can see a DEFINE\_COM

definition has been added for the web browser control:

DEFINE\_COM CLASS(#VA\_WEBCTL.WebBrowser) NAME(#VA\_WEBCT The methods and events and the Visual LANSA properties of the control are now accessible in the Details tab.

From the Features window (F2) you can display a brief description of all the properties, events and methods of the control.

Note that the control has a Navigate method which displays a URL:



Add this statement to the initialize event of your form:

invoke method(#VA\_WEBCTL.navigate) url(www.lansa.com)

To see how the browser control works, compile the form and execute it. The LANSA home page is displayed in the browser.

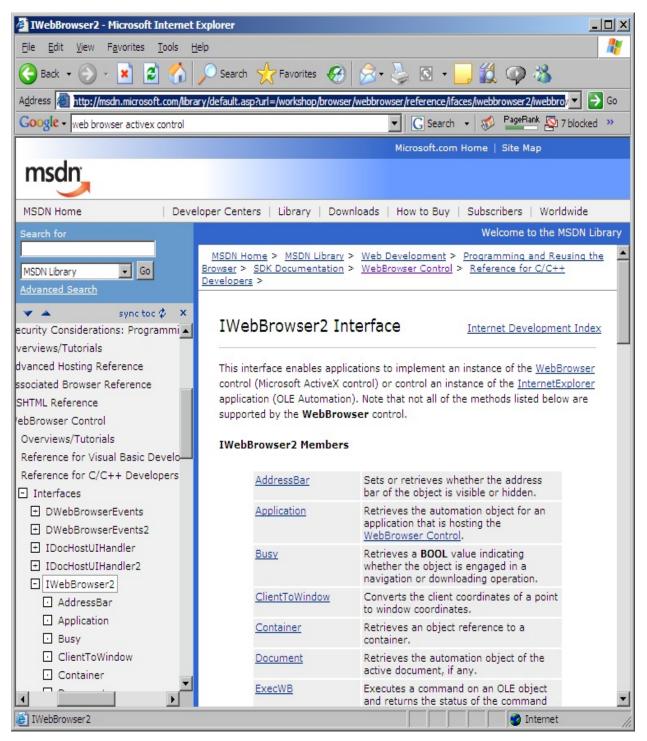


## 17.1 Using ActiveX Controls

#### **Setting Properties at Design Time**

You use the properties of an ActiveX control exactly the same way as you use the properties of a native LANSA control.

Note that with the web browser control we are using, some of the properties listed do not have any effect on the browser. This is because they are Internet Explorer properties only. For example the AddressBar property has no effect in the web browser because by definition it does not have an address bar.



Set the width of the browser control by using its Width or Width\_COM property (they are identical):

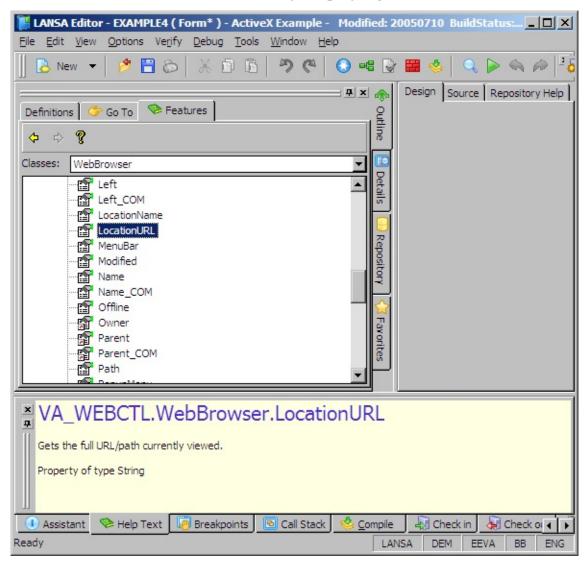
VisualStyleOfParent	True	
Width	400	
Width_COM	400	
		<b>_</b>

↑ 7.1.7 Using an ActiveX Control

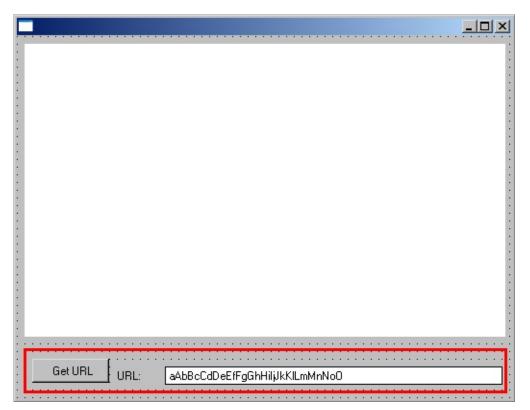
## **Retrieving the URL from the Browser Control**

This section shows you how you can retrieve the value of a property from the browser control.

The browser control has a property LocationURL which retrieves the URL of the resource that the browser is currently displaying:



To use this property, add a push button and field to your form to retrieve the URL:



Set the properties of the button and the field:

Push button control caption('Get URL')

Field STD\_TEXTS caption('URL:') labeltype(Caption) marginleft(50)

In the Click event of the push button, add this code to retrieve the URL: EVTROUTINE HANDLING(#PHBN\_1.Click) change field(#std\_texts ) to(#VA\_WEBCTL.LocationURL) ENDROUTINE

Compile and execute the form. When the LANSA page is displayed in the browser control click on the Get URL button to see the URL:

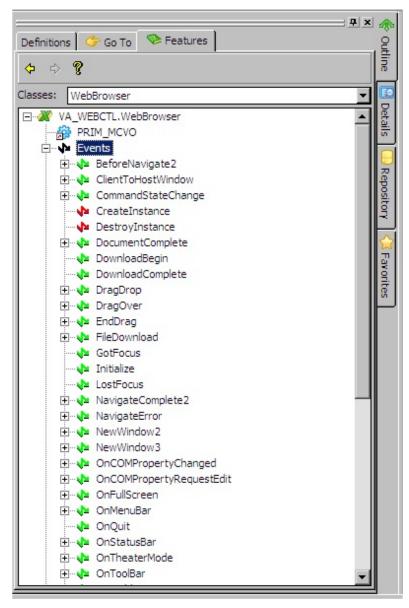


↑ 7.1.7 Using an ActiveX Control

#### **Events**

This section shows how to use the events of an ActiveX control.

To see which events are available for the browser control display its Features (F2) and expand the list of its events.



The browser control has a DocumentComplete event which is triggered when the page has been found and downloaded. You can use this event to show a message when the page download is complete. Add this code to your form:

EVTROUTINE HANDLING(#VA\_WEBCTL.DocumentComplete) OPTIONS use builtin(MESSAGE\_BOX\_SHOW) with\_args(OK OK Information 'Page R

#### ENDROUTINE

Compile and execute the form. The completion message is displayed when the LANSA page has been downloaded:



↑ 7.1.7 Using an ActiveX Control

## A Note about Initializing ActiveX Controls

Note that visual ActiveX controls are not fully realized when the Visual LANSA CreateInstance Event is signalled. Non-visual components do not cause a problem.

This means that accessing any feature of a Visual ActiveX control in a CreateInstance event routine may cause runtime errors. Use Initialize instead.

↑ 7.1.7 Using an ActiveX Control

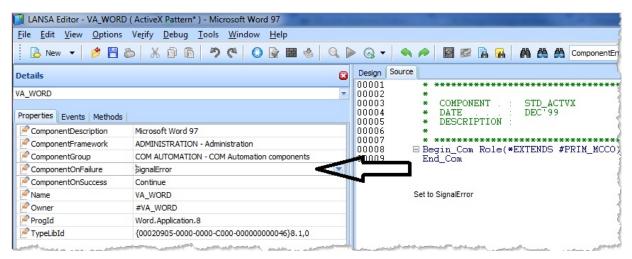
## 7.1.8 ActiveX Error trapping

Following is an example to explain the concept of ActiveX Error trapping. Copy and paste the code into a RDMLX enabled form, compile and run it.

To gain an understanding of the concept, follow the comments within the supplied code.

How to set it up:

- 1. Set the ActiveX ComponentOnFailure Property.
- 2. Trap error in code.
- 3. Set the ActiveX ComponentOnFailure property to "SignalError".



#### \*Trap error in code

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(376) Clientwidth(208) Height(414) Left(295) Top(178) Width(224)

Define\_Com Class(#PRIM\_PHBN) Name(#StartWord) Caption('Start Word') Displayposition(1) Left(8) Parent(#COM\_OWNER) Tabposition(1) Top(8) Width(193)

Define\_Com Class(#PRIM\_PHBN) Name(#CreateDocument) Caption('Create Document') Displayposition(2) Left(8) Parent(#COM\_OWNER) Tabposition(2) Top(40) Width(193)

Define\_Com Class(#PRIM\_PHBN) Name(#FindBookmark) Caption('Access Bookmark') Displayposition(3) Left(8) Parent(#COM\_OWNER) Tabposition(3) Top(72) Width(193)

Define\_Com Class(#va\_word.Application) Name(#Word) Reference(\*dynamic) Define\_Com Class(#va\_word.Document) Name(#ActiveDocument) Reference(\*dynamic) Define\_Com Class(#va\_word.Bookmarks) Name(#Bookmarks) Reference(\*dynamic)

Evtroutine Handling(#StartWord.Click)

\* Start an instance of word Set\_Ref Com(#Word) To(\*Create\_as #va\_word)

Endroutine

Evtroutine Handling(#CreateDocument.Click)

If (#Word \*IsNot \*null)

If (#ActiveDocument \*Is \*null)

\* **Open a new document** #ActiveDocument <= #Word.Documents.Add

\* Get the reference to the Bookmarks of the Active Document \* This is necessary to be able to listen to the activeX errors #Bookmarks <= #ActiveDocument.Bookmarks

Else

Use Builtin(ov\_message\_box) With\_Args("Document Aleady Exists")

Endif

Else

Use Builtin(ov\_message\_box) With\_Args("Word Has Not Yet been Started")

Endif

Endroutine

Evtroutine Handling(#FindBookmark.Click)

Define\_Com Class(#va\_word.Bookmark) Name(#Bookmark) Reference(\*dynamic)

\* Ensure there is an Active Document If (#ActiveDocument \*IsNot \*null)

\* Try to access a bookmark

\* As this is a new document there will be no bookmarks

\* This will generate an ActiveX error on Bookmarks ONLY

#Bookmark <= #Bookmarks<1>

Else

Use Builtin(Ov\_message\_box) With\_Args("There Is No Active Document")

Endif

Endroutine

Evtroutine Handling(#Word.ComponentError)

# \* This routine will never fire. The bookmark error is specific to the Bookmark instance

Use Builtin(Message\_Box\_Add) With\_Args("Error on Word") Use Builtin(Message\_Box\_Add) With\_Args(#Com\_err\_info.ErrorWord)

Use Builtin(Message\_Box\_Show)

Endroutine

Evtroutine Handling(#Bookmarks.ComponentError)

Use Builtin(Message\_Box\_Add) With\_Args("Error on Bookmarks") Use Builtin(Message\_Box\_Add) With\_Args(#Com\_err\_info.ErrorWord)

Use Builtin(Message\_Box\_Show)

Endroutine

Evtroutine Handling(#Com\_Owner.Closing)

\* **Drop all references and ensure Word quits correctly** #Word.Documents.Close

If (#Word \*IsNot \*null)

#Word.Quit

Endif

Endroutine

End\_Com

#### **Background notes**

To handle runtime errors in LANSA, we need to understand the difference between LANSA generated errors and errors generated by ActiveX controls.

Consider the following code sample:

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Clientheight(313) Clientwidth(492) Define\_Com Class(#PRIM\_STBR) Name(#STBR\_1) Displayposition(1) Height(24) Left(0) Messageposition(1) Parent(#COM\_OWNER) Tabposition(1) Tabstop(False) Top(278) Width(484) Define\_Com Class(#VA\_WORD.Application) Name(#VA\_WORD) Evtroutine Handling(#com\_owner.Initialize) Message Msgtxt("Throwing ActiveX error") #VA\_WORD.ComponentOnFailure := SignalError
If (#VA\_WORD.ActiveDocument.Bookmarks.Exists( " ERROR" ))
Endif
Endroutine

Evtroutine Handling(#VA\_WORD.ComponentError) Options(\*NOCLEARMESSAGES \*NOCLEARERRORS) Message Msgtxt(("Catching ActiveX error").BlankConcat( #COM\_ERR\_INFO.ErrorCode.AsString )) #COM\_ERR\_INFO.Clear Endroutine End\_Com

In the above example, the initialize event references

Word.ActiveDocument.Bookmarks. However, if there is no ActiveDocument, that is, you have neither explicitly created nor opened a document, the ActiveDocument reference in Word will be null. This means that the moment the Visual LANSA runtime attempts to access the ActiveDocument property of Word, there will be a LANSA generated runtime error reporting that Word.ActiveDocument is Null.

Had the initialize event been as follows, no such error would occur.

```
* Add a new (active) document.
#VA_Word.Add
If (#VA_word.ActiveDocument.Bookmarks.Exists("Etc")
etc.
```

It is therefore the responsibility of the developer to ensure that before trying to access a feature that references another instance, that suitably defensive code is included.

```
If (#Word.ActiveDocument *isnot *null)
If (#VA_word.ActiveDocument.Bookmarks.Exists("Error")
etc.
```

else

Use OV\_message\_box ("There is no Active Document")

Endif

Even though this code now avoids the LANSA generated runtime error, you still can't detect an ActiveX error by listening to VA\_WORD because the ActiveX error for a missing bookmark belongs to the Bookmarks feature. Therefore, you need to set up specific references in the code.

Define\_Com Class(#va\_word.Application) Name(#Word) Reference(\*dynamic) Define\_Com Class(#va\_word.Document) Name(#ActiveDocument) Reference(\*dynamic) Define\_Com Class(#va\_word.Bookmarks) Name(#Bookmarks) Reference(\*dynamic)

Evtroutine #Com\_owner.CreateInstance

```
* Start Word
Set_Ref Com(#Word) To(*Create_as #va_word)
```

```
* Create a new document and store the reference
#ActiveDocument <= #Word.Documents.Add
#Bookmarks <= #ActiveDocument.Bookmarks
```

Endroutine

When you now attempt to access a bookmark that doesn't exist, the error generated by Word can be detected as follows:

Evtroutine Handling(#Bookmarks.ComponentError)

Use Builtin(Message\_Box\_Add) With\_Args("Error on Bookmarks") Use Builtin(Message\_Box\_Add) With\_Args(#Com\_err\_info.ErrorWord)

Use Builtin(Message\_Box\_Show)

Endroutine

↑ 7.1.7 Using an ActiveX Control

## 7.1.9 Licensing and Deployment

When you purchase an ActiveX control, you buy a design-time license for the control. This license is installed on your PC when you install the product and it allows you to use the control in the applications you are developing. The products typically have various design-time licenses for sale. These range from single developer licenses to multiple seat and network licenses.

Usually when you buy a design-time license you also get permission to distribute unlimited copies of the control with your application as compiled object code.

When you distribute your application that uses an ActiveX control, you need to ship also the control in binary format (usually a .dll or an .ocx). Do not distribute the setup file that comes with the design-time license.

#### **Design-time license**

If a control has a design-time license it will usually install a license key when its setup program is run. The license key is usually entered into the registry although it can also be installed in a special file.

Having a design-time license allows you to use the control for developing applications.

#### **Run-time license**

When you have built and compiled an executable application that uses the control, you need to distribute this control together with the application so a customer can use it. Most of controls also support run-time licensing for this purpose.

At compile time a license key is embedded into the executable file and this key is then inserted into the control when it is created. The control checks this key and if it is valid the control starts up. Because a key is embedded into the executable, no license key is required to be installed on the customer's machine and only the ActiveX control's binary needs to be distributed.

↑ 7.1 Using ActiveX Controls

## 7.2 Integrating with ActiveX-Enabled Applications

In addition to using ActiveX controls in your LANSA application, you can also use ActiveX technology to integrate your application with other ActiveXenabled applications. Examples of ActiveX-enabled applications are, for instance, Microsoft Office applications, Lotus Notes and Crystal Reports.

You can manipulate programmatically the objects inside ActiveX-enabled applications from within your LANSA application. For example when you are using Microsoft Word from your LANSA application, you can directly manipulate any of Word's objects (for instance Window, Document, Paragraph) and their properties (such as Name, PageSetup), methods (CheckSpelling, Print) and events (for example New, Open).

In this section we show how to integrate a LANSA application with Microsoft Word version 9 (Word 2000). The same principles described here apply equally when you are integrating other ActiveX-enabled applications with LANSA.

- 7.2.1 Enrolling an ActiveX-Enabled Application
- 7.2.2 Accessing the Application's Objects
- 7.2.3 Important Note about Syntax with ActiveX Collections
- 7.2.4 Example of Integrating Microsoft Word
- 7.2.5 Properties of Type IDispatch
- ↑ 7. ActiveX Controls

# 7.2.1 Enrolling an ActiveX-Enabled Application

As with ActiveX controls, you need to enrol ActiveX-enabled applications in the LANSA repository as a component before you can use them from your LANSA application.

This section shows you how to enrol an ActiveX-enabled component in the repository using Microsoft Word as an example. To use Microsoft Word through ActiveX you can also use the pre-enrolled component VA\_WORD which is shipped with LANSA.

To enrol Microsoft Word application in the repository, create an ActiveX component Word in the ActiveX components group:

😻 New activeX		×
Name	WORD	Create
Description	Word Integration	
Initial public access	Normal	Ca <u>n</u> cel
Framework Administration Common Control Component Customer Service C dfdgh Executive Informat Budgeting & Accour Personnel & Payrol Latornal System Co	omponents on Systems ting	
	up Boxes n Buttons	

Open the ActiveX Component Browser by clicking on the button with three dots in the ProgID property.

ActiveX Component Browser				
l				
<u> </u>				

Expand the Other Components list. Scroll down to the Microsoft Word components. Notice that there are components for Microsoft Word versions 2 and 6. Word 2000 (Version 9) components are called Word Application and Word Document, without a version number.

ActiveX Component Browser	×
⊕- Microsoft Windows Installer	
Microsoft Windows Installer Message RPC	
Microsoft Word 2.0 Document	
. Microsoft Word 6.0 - 7.0 Document	
. Microsoft Word 6.0 - 7.0 Picture	
Microsoft Word Application	
Microsoft Word Basic	
Microsoft Word Document	
Microsoft Word Picture	
. Microsoft WordArt 1.0	
. Microsoft WordArt 2.0	
Microsoft Works AutoCorrect Service	
庄 - Microsoft Works Calendar	
Microsoft Works Calendar DIExports	
. Microsoft Works Font Cache	
. Microsoft Works Html Print Service	
Histore Huppelink Service	الشر ،
	<u> </u>
<u>K</u> Cancel	

The Application component represents the Word application, the Document component represents a Word document (the document can also be accessed

through the application). Enrol the Word Application.

Click OK to specify the ProgID. Notice that the TypeLibId is not filled in. (Whenever possible, the type library information is filled in automatically by LANSA.) If you now display the Source view of the component, you notice that the BEGIN\_COM statement shows an error telling the type library has not been specified.

Tegin\_com role(\*EXTENDS #PRIM\_MCCO) progid('Word.Application.9') typelibid('')
 Operation failed with a HRESULT of 0x80029C4A. Component with ProgId of "Word.Application.9" requires a Type Library to be specified.
 END\_COM

To specify the type library, click on the TypeLibId property in the Details tab and then click the button with three dots. The ActiveX Type Library Browser window is displayed. Select the Microsoft Word 9.0 Object Library as the type library.

ActiveX Type Library Browser	×
Microsoft Visual Studio .NET VB and C# Project Model - version 7.0	
🗄 🗄 Microsoft Windows Common Controls 5.0 (SP2) - version 1.3	
🗄 🗄 Microsoft Windows Common Controls 6.0 (SP6) - version 2.0	
🗄 🗄 Microsoft Windows Common Controls-2 5.0 (SP2) - version 1.1	
🗄 🗄 Microsoft Windows Common Controls-2 6.0 (SP4) - version 2.0	
🗄 🗄 Microsoft Windows Common Controls-3 6.0 (SP4) - version 1.1	
🗄 🗄 Microsoft Windows Image Acquisition 1.01 Type Library - version 1.0	
🗄 Microsoft Windows Installer Object Library - version 1.0	
Microsoft WMI Scripting V1.2 Library - version 1.2	
Microsoft Word 9.0 Object Library - version 8.1	
🗄 Microsoft Works Calendar 6.0 Type Library - version 6.0	
🗄 🕀 Microsoft Works GDI Cache 5.0 Type Library - version 5.0	
庄 - Microsoft Works Help 6.0 Type Library - version 6.0	
庄 Microsoft Works Launcher 5.0 Type Library - version 5.0	
庄 Microsoft Works Print Service 5.0 Type Library - version 5.0	
Microsoft ) (orke Ouill Control 5.0 Tupo Library Lorsion 5.0	- <b>1</b>
	-
QK Cancel	

Save the component.

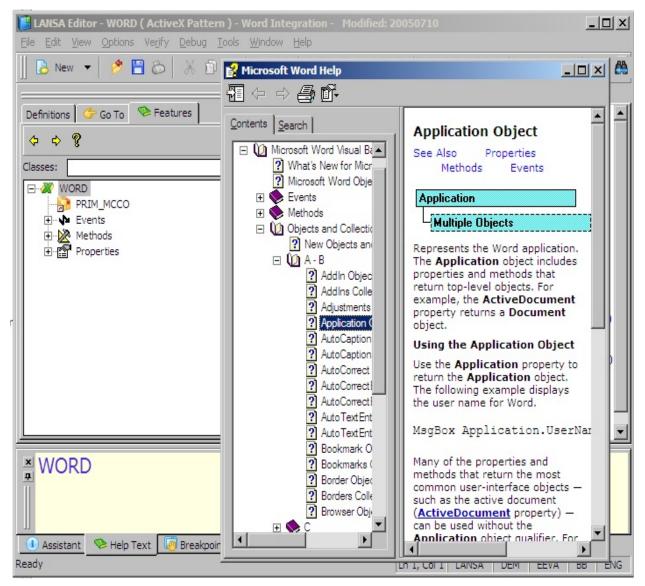
↑ 7.2 Integrating with ActiveX-Enabled Applications

## 7.2.2 Accessing the Application's Objects

There are usually only a limited number of properties and methods of the ActiveX-enabled application itself (such as Microsoft Word) you need to work with. You will mainly be working with objects inside the application. You access these objects via the application.

To do this you need to know the object hierarchy of the ActiveX-enabled application you are using from LANSA. Usually this hierarchy is described in the online documentation of the application itself.

To see the Word online documentation, select the Word Application in the Features tab and press F1:



MS Word Application contains a comprehensive set of objects:

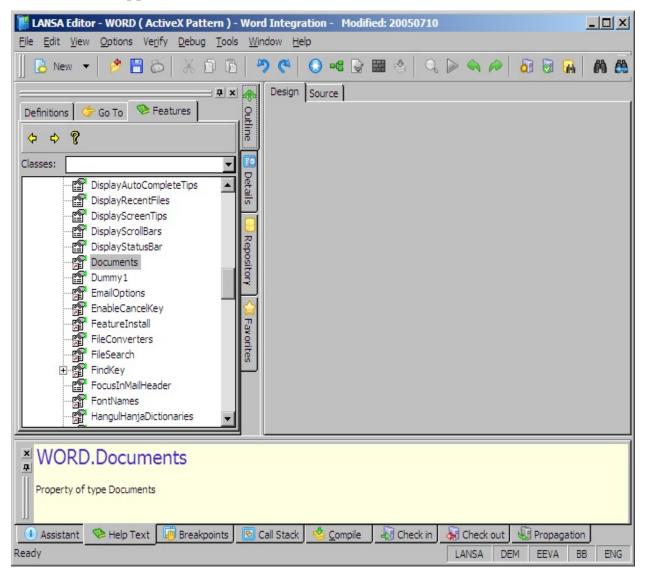
😫 Microsoft Word Help		
┨ ← ⇒ <b>⊜</b> ि		
Contents       Search	Microsoft Word Objects See Also Application AddIns (AddIn)	KeysBoundTo (KeyBinding)
<ul> <li>Methods</li> <li>Objects and Collections</li> <li>Properties</li> <li>Microsoft Office Visual Basic</li> <li>Microsoft Forms Visual Basic</li> </ul>	AnswerWizard Assistant AutoCaptions (AutoCaption) AutoCorrect Browser	<ul> <li>KeyBindings (KeyBinding)</li> <li>Languages (Language)</li> <li>Dictionaries (Dictionary)</li> <li>LanguageSettings</li> <li>ListGalleries (ListGallery)</li> </ul>
	CaptionLabels (CaptionLabel)     COMAddIns (COMAddIn)     CommandBars (CommandBar)     CommandBarControls (CommandBarControl)	ListTemplates (ListTemplate ListLevels (ListLevel) MailingLabel CustomLabels (CustomLabe
	<ul> <li>DefaultWebOptions</li> <li>Dialogs (Dialog)</li> <li>Dictionaries (Dictionary)</li> <li>Documents (Document)</li> </ul>	- MailMessage - Options - <mark>RecentFiles (RecentFile)</mark> - Selection
	EmailOptions EmailSignature FileConverters (FileConverter)	- <mark>SpellingSuggestions (SpellingS</mark> - SynonymInfo - System
	FileSearch FontNames HangulHanjaConversionDictionaries (Dictionary)	Tasks (Task) Templates (Template) VBE Windows (Window) ▼

↑ 7.2 Integrating with ActiveX-Enabled Applications

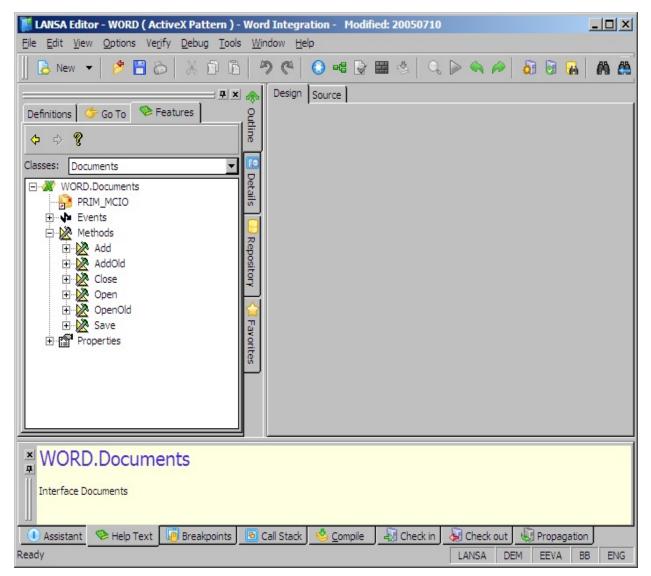
# **Documents Collection**

Documents is the type of object most commonly used in Microsoft Word. The Word application contains a document's collection object that consists of all documents currently open in Word and a document object.

To see the Documents property of the Word application, open the Features help for the Word application.



Double-click the Document's property to display the Document's collection. You can use the methods of the document's collection to create new documents (Add method).



You refer to members of the document collection (individual documents) by specifying the document's key. This key can be either the name of the document or its number (documents are numbered in the order you create or open them.) This is the syntax:

invoke method(#wordapp.documents.item<'test.doc'>.activate)

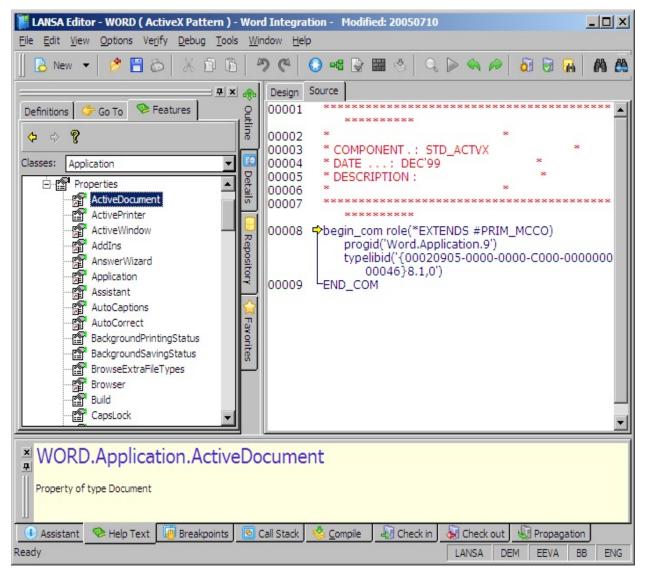
or

invoke method(#wordapp.documents.item<1>.activate)

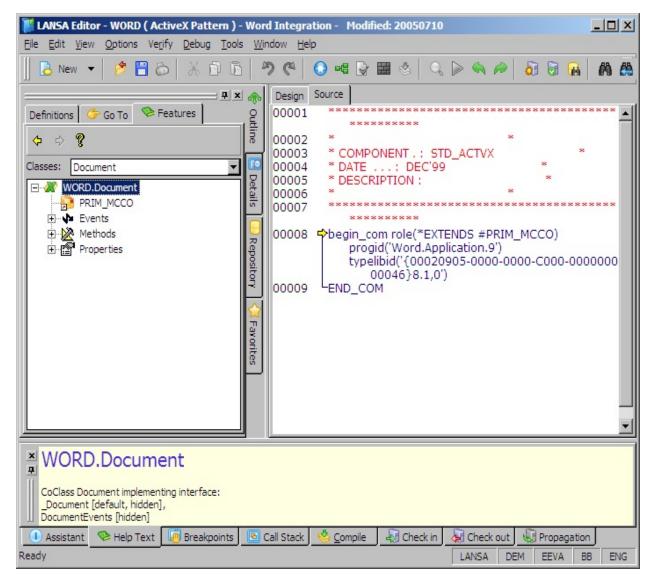
↑ 7.2.2 Accessing the Application's Objects

# **ActiveDocument Property**

The Word application object contains a property ActiveDocument which is the currently active document.



The ActiveDocument property refers to the document object. Double-click on the ActiveDocument property to display the Document object and its properties, events and methods.



You use the document properties and methods for example to name documents, save and print them.

↑ 7.2.2 Accessing the Application's Objects

#### 7.2.3 Important Note about Syntax with ActiveX Collections

When you work with a collection in an ActiveX component using RDMLX you must use the RDMLX item keyword. For example, the Word VBA statement to activate a document:

Documents("test.doc").activate

Becomes in RDMLX:

invoke method(#wordapp.documents.item<'test.doc'>.activate)

Note also that instead of:

- Parenthesis for the item in the collection you use the greater than and less than signs <>.
- Double-quotes you use single quotes.
- ↑ 7.2 Integrating with ActiveX-Enabled Applications

### 7.2.4 Example of Integrating Microsoft Word

In this example, Microsoft Word is integrated with LANSA using ActiveX. The example application can be used to start Word, create new documents based on a template, insert data from the repository to the document and to close, print and save documents.

Microsoft Word ActiveX	ixample	
Word Application	Data for New Document	-Work with Document
Start Word	Double-click an employee to insert data.	Document1
Show/Hide Word	Employee Gi Employee  JOHN SMYTHE Robert SMITHE Robert	
Word WindowState O Minimized O Normali	PAUL SMITHS PETER SMITHS JACK SMITHE GEORGE SNELL ALLAN SNEDDON	
O Maximized	DAMIAN SNASHA WILLIAM PERRY CHRISTOP PERRIN PATRICK PAUL	Save
Add Document The document will be based on the Word	GEORGE PATTISON JOHN MOORE BRADLEY WOODS	Close
template vI_sam099.dot installed by Partition Initialization.	JACK TURNER GARY NEAVE PAUL ZACHARIA CHARLES DICKENS ADAM PET DOUGLAS	
Exit Word	DAVID MCCULLY KELLY THOMP DAVID REID V	

You can copy and paste the example from Example Source .

In the following we use the pre-enrolled component #VA\_WORD as the Microsoft Word ActiveX control.

↑ 7.2 Integrating with ActiveX-Enabled Applications

### **Referencing the Application**

Start Word

To start an ActiveX-enabled application from a LANSA form, you need to add a reference to it. The simplest way of doing this is to drag and drop the ActiveX component to the form. This creates a DEFINE\_COM statement for the component.

This is the statement created for the Word application component when the #VA\_WORD component is dropped to a form:

DEFINE\_COM CLASS(#VA\_WORD.Application) NAME(#VA\_WORD)

When you execute your form and this DEFINE\_COM statement for Word is encountered, an instance of the Word application is started immediately. Even if you have an instance of Word running, a new instance of it is started.

Change the name of the application to #WordApp to make the example easier to follow:

DEFINE\_COM CLASS(#VA\_WORD.Application) NAME(#WordApp)

### **Controlling when the Reference is Created**

In this example, the Word application is not to start when the LANSA form is executed. To control when Word is started, first define it with a dynamic reference and then assign a reference to it in the Click event of a button.

To make the component definition dynamic, add the Reference property to the DEFINE\_COM statement like this:

DEFINE\_COM CLASS(#VA\_WORD.Application) NAME(#WordApp) REFE

When this statement is executed, no reference to Word is created. To start Word when a user clicks on the Start Word button, assign a reference to it in the button's Click event using the SET\_REF command:

EVTROUTINE HANDLING(#STARTBTN.Click)

SET\_REF COM(#WordApp) TO(\*CREATE\_AS #VA\_WORD.application) ENDROUTINE

### **Checking if the Reference is Exists**

Trying to use any of Word's properties, methods or events before it is started would cause an error. Therefore, initially disable the controls on the form and enable them only when Word has been started. The IF\_REF statement is used to check whether a reference has been created to Word. Put this statement in the Click event of the Start Word button:

EVTROUTINE HANDLING(#STARTBTN.Click) SET\_REF COM(#WordApp) TO(\*CREATE\_AS #VA\_WORD.application)

if\_ref com(#wordapp) is\_not(\*null)

set com(#ShowHideBtn #gpbx\_1 #rdbn\_1 #rdbn\_2 #rdbn\_3 #Addbtn #InsB endif

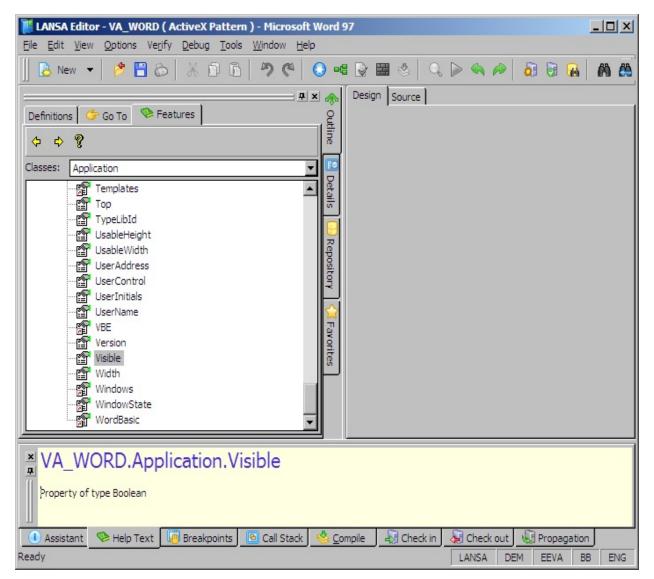
ENDROUTINE

### **Setting Properties**

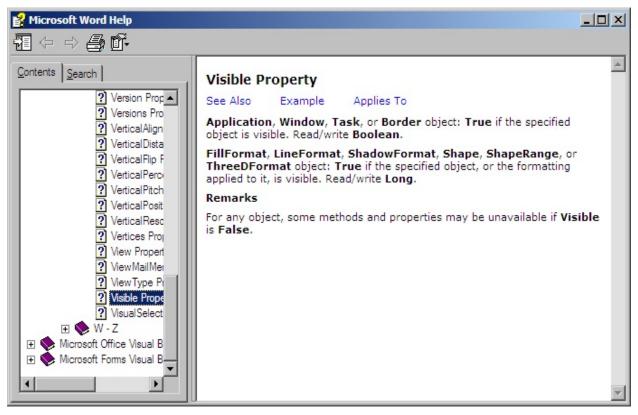


#### **Boolean Property**

The Visible property of the Word application controls whether the Word window is visible. Use the Features Help (F2) to see what type of value this property accepts.



If you want more detailed information, press F1 to open the help for this property. The help is displayed only if Visual Basic Help was included in your Word or Office installation.



Make the Show/Hide Word button toggle the Visible property on or off:

EVTROUTINE HANDLING(#ShowHideBtn.Click)

if cond('#wordapp.visible \*eq True')

SET #WordApp Visible(false)

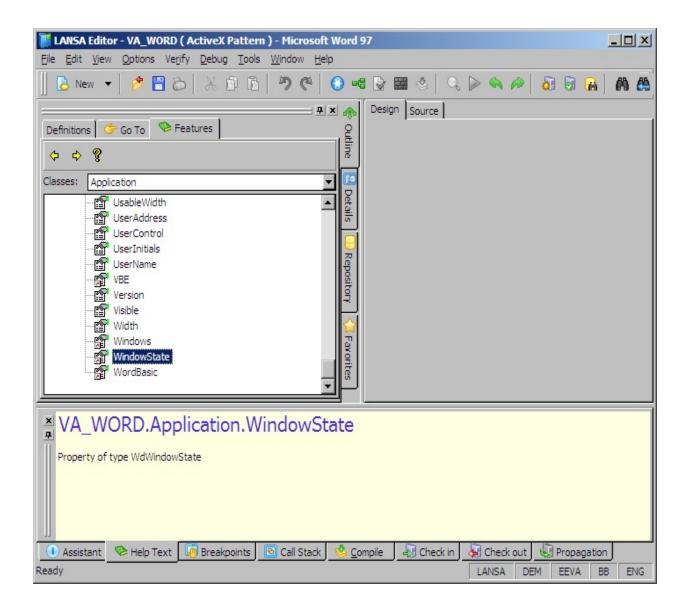
else

SET #WordApp Visible(true)

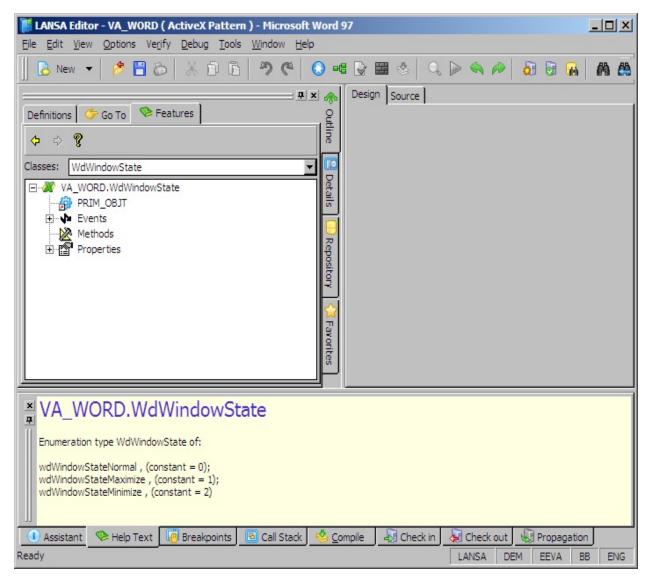
endif ENDROUTINE

#### **Enumeration Property**

To let the user set the size of the Word window, use the WindowState property of the application. The Features help for the WindowState property shows that the value for this property must be of the type WdWindowState.



Double-click the WindowState property to see what values you can specify for it.



The property can have one of three constants (enumerations) as the value. Note that you must qualify the enumeration name by the type library name VA\_WORD. Three radio buttons were used for setting the window state:

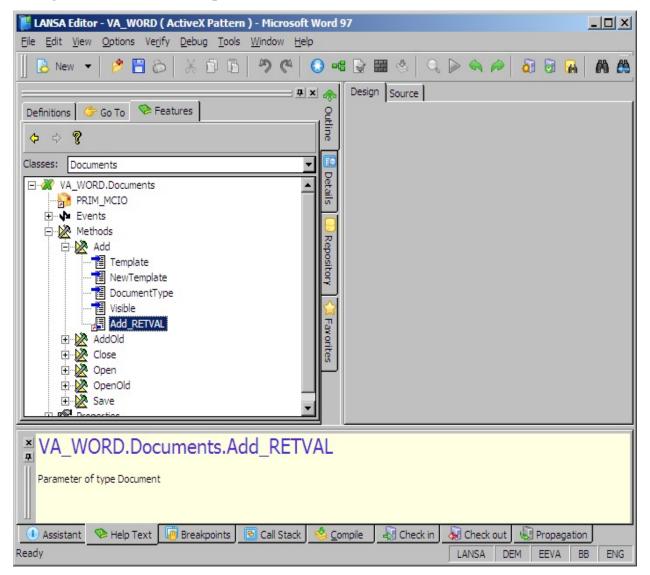
```
EVTROUTINE HANDLING(#RDBN_1.Click)
IF COND('#rdbn_1.buttonchecked *eq true')
SET #WordApp Windowstate(VA_WORD.wdWindowStateMinimize)
endif
ENDROUTINE
EVTROUTINE HANDLING(#RDBN_2.Click)
if cond('#rDbn_2.buttonchecked *eq True')
SET #WordApp Windowstate(VA_WORD.wdWindowStateNormal)
endif
ENDROUTINE
```

EVTROUTINE HANDLING(#RDBN\_3.Click) if cond('#rdbn\_3.buttonchecked \*eq True') SET #WordApp Windowstate(VA\_WORD.wdWindowStateMaximize) endif ENDROUTINE

### Using the Add Method

Add Document

To create a new document, use the Add method of the documents collection of the Word application. In this method you can specify the Word template on which the new document is based on. The method returns a reference to the newly created document. You assign this return value to a component name using the Add\_RETVAL parameter.



In order to use the Add\_RETVAL parameter to assign a component reference to the new document, the component has to first be defined. This component must be of the type Word document and it must be defined with a dynamic reference:

DEFINE\_COM CLASS(#VA\_WORD.Document) NAME(#WORDDOC) REF

In this example, the Add method in the Click event of the New Document button is invoked. The new document will be based on the vl\_sam99.dot template which is installed as part of the LANSA demonstration system. Give the new document the component name #Worddoc:

EVTROUTINE HANDLING(#ADDBTN.Click) invoke method(#WordApp.documents.add) template('C:\X\_Win95\x\_lansa\x\_c set #std\_num value(#WordApp.Documents.count) ENDROUTINE

Assigning a component name to the new document can be useful. However, there are also other ways of identifying a document. You can, for example, refer to the currently active document (#WordApp.ActiveDocument) or you can identify it by its name (#WordApp.Documents<'Report.doc'>) or number (#WordApp.Documents<2>) in the documents collection.

### **Using Methods to Insert Text**

Our example application inserts text from the LANSA repository for a selected employee to the Word document. The places where text will be inserted in the document in are marked as bookmarks in the document. To view them, create a new document based on the vl\_sam099.dot template and select the Bookmarks option from the Insert menu in Word.

Bookmark	? ×
Bookmark name:	
FULLNAME	
FULLNAME ADDRESS1 ADDRESS2 ADDRESS3 POSTCODE PHONEHME PHONEBUS SKILLS HOMEPAGE	▲ 
Sort by: C Name C Location	
Hidden bookmarks	
<u>A</u> dd <u>D</u> elete <u>G</u> o	То
C	ose

The bookmarks are shown as brackets in the document:

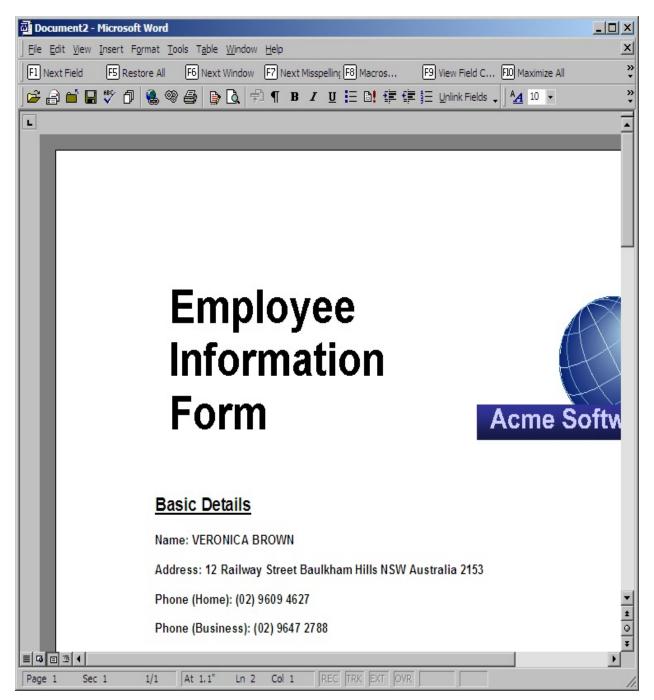
👜 Document3 - Microsoft Word		
Elle Edit View Insert Format Iools Table Window Help		
FI Next Field F2 Move Text F5 Restore All F6 Next Window F7 Next Misspe		
Ĵ☞ @ ╈ ∎ ♥ Ӣ 🝓 ☜ @ ┣ ┣ ╡ ¶ в и ⊻ Ε Ϸ! ╪ ჼ		-
Employee Information Form		Acme Software
Basic Details          Name:         Address:       ][][][]         Phone (Home):       ]		
Phone (Business):		
Skills		
Skill Description	Grade	Comments
Code		
		ee Information Site.
(This page was created	on	by[_])
Page 1 Sec 1 1/1 At 3.4" Ln 1 Col 1 REC TRK EXT OV	R	

All the bookmarks in the document make up its Bookmarks collection. You refer to the individual bookmarks by their name:

#WordDoc.Bookmarks.item<'address1'>

The Insert Data for Employee button is used to insert the details of the currently selected employee in the document. In the Click event of this button a bookmark is first selected using its Select method and the current contents of a LANSA field are then inserted using the TypeText method:

invoke method(#WordDoc.Bookmarks.item<'address1'>.select)
invoke method(#WordApp.Selection.TypeText) text(#address1)



These methods are invoked for all the bookmarks in the document.

### **Print Method**

Print

The Print method of the Word document is used to print it: invoke method(#WordDoc.PrintOut)

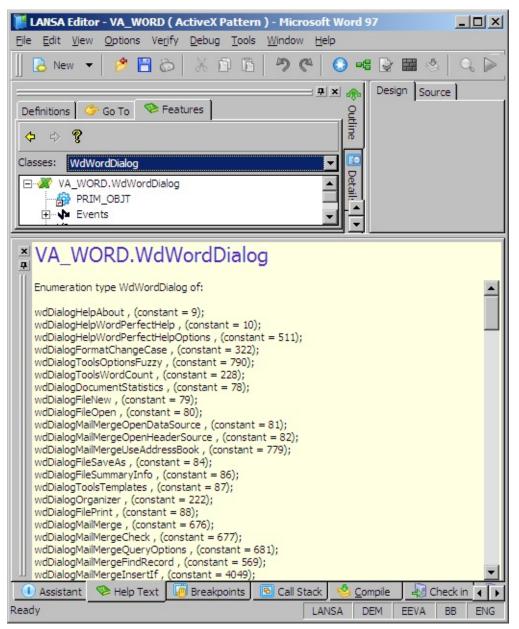
If you are not using a component reference to the document, you can, for example, identify the document as the currently open document in the Word application:

invoke method(#WordApp.ActiveDocument.PrintOut)

### **Showing a Word Dialog**

Save...

The Save... button in our example displays the Word built-in Save Document dialog using the Show method of the Dialog object. You can access any Word built-in dialog using this method.



To access the Word built-in dialogs you need to use the ITEM keyword and you need to enclose the name of the dialog in <> signs. The name of the dialog has to be qualified by the name of the type library, VA\_WORD:

invoke method(#WordApp.Dialogs.item<VA\_WORD.wdDialogFileSaveAs>.s

4	Save As					
	Save in:	🕪 Local Disk (C:)	💌 🗢 🗈 🔍 🗡	III - Tools -		
1		@LANSADOC	i 1386	Program Files		
I		a	Cons	RECYCLER		
I	History	August spin	Cicons2	🚞 set		
I		b	🛅 Interface Redesign	🚞 System Volume Informatic		
I		Copy (2) of admin	internal_systems	🗀 Usability		
I		Copy of admin	C Iocomp	C WINDOWS		
I	My Documents	DELL	🚞 Jason	🖳 ~\$oduct Status 2005061		
I		DeveloperGuide	🛅 lj 1200	autohide.doc		
I		Documents and Settings	C MSOCache	COMPONENT.doc		
I		Download	My Documents	Doc1.doc		
I	Desktop	DRIVERS	C My Music	Doc5.doc		
I		EEVA	C New Folder	docking.doc		
I		Frameworks	Dew Folder (2)	eevastatus.doc		
I	Favorites	HELP	Outlook	eevatasklist.doc		
I	ravorites	i HelpRewrite	photos	EevaTasks.doc		
I				•		
l	My Network	File <u>n</u> ame: Basic Details.doc		Save		
	Places	Save as type: Word Document (*	.doc)	▼ Cancel		

The functionality of the Save As dialog is now available to you:

# **Getting a Return Value**

The Show method of the Dialog object has two parameters: TimeOut and return value (Show\_RETVAL):

📕 LANSA Editor - VA_WORD ( ActiveX Patter	n) - Microsoft	Word 9	7	_ 🗆 🗵
Eile Edit View Options Verify Debug Tools	s <u>W</u> indow <u>H</u> elp			
📗 🔁 New 🔻 🛛 🤌 🔚 🔂 🛛 🔏 🗊 🖻	56	0 =8	2	\$ 0°;
Definitions Go To Features	₽ 3	Coutline Setails Repository C Favorites	Design	Source
VA_WORD.Dialog.Show				
🚺 Assistant 👒 Help Text 🔲 Breakpoints	🖻 Call Stack	🍓 <u>C</u> om	npile 4	Check 🖌 🕨
Ready	LANSA	DEM	EEVA	BB ENG

The return value parameter indicates whether the OK (1) or the Cancel (0) button was clicked. This code shows a message if the OK button was clicked on the SaveAs dialog box:

```
invoke method(#WordApp.Dialogs.item<VA_WORD.wdDialogFileSaveAs>.s
if cond('#std_num *ne 0');
execute infmessage 'Document has been saved.';
endif;
```

### **Example Source**

Here is the source code for the Word integration example:

\*;

FUNCTION OPTIONS(\*DIRECT) BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) CAPTION('Microsoft Wo DEFINE\_COM CLASS(#VA\_WORD.Application) NAME(#WordApp) REFE DEFINE\_COM CLASS(#VA\_WORD.Document) NAME(#WORDDOC) REF

DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#STARTBTN) CAPTION('S DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#ADDBTN) CAPTION('Add DEFINE COM CLASS(#PRIM LTVW) NAME(#LTVW 1) DISPLAYPOSI DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_1) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_2) DISPLAYPOSIT DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#PRINTBTN) CAPTION('Pi DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_3) PARENT(#LTVV DEFINE COM CLASS(#PRIM LVCL) NAME(#LVCL 4) PARENT(#LTVW DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_5) PARENT(#LTVW DEFINE COM CLASS(#PRIM LVCL) NAME(#LVCL 6) PARENT(#LTVW DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_7) PARENT(#LTVW DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_8) PARENT(#LTVW DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#SAVEBTN) CAPTION('Sa' DEFINE\_COM CLASS(#PRIM\_GPBX) NAME(#GPBX\_1) CAPTION('Word DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#RDBN\_1) CAPTION('Min DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#RDBN\_2) BUTTONCHEC DEFINE\_COM CLASS(#PRIM\_RDBN) NAME(#RDBN\_3) CAPTION('Max DEFINE\_COM CLASS(#PRIM\_GPBX) NAME(#GPBX\_2) CAPTION('Word DEFINE\_COM CLASS(#PRIM\_GPBX) NAME(#GPBX\_3) CAPTION('Data DEFINE\_COM CLASS(#PRIM\_GPBX) NAME(#GPBX\_4) CAPTION('World CLASS(#PRIM\_GPBX) NAME(#GPBX NAME(#GPBX) NAME(#GPBX) NAME(#GPBX) NAME(#GPBX\_4) CAPTION('World CLASS(#PRIM\_GPBX) NAME(#GPBX\_4) NAME(#GPBX\_4 DEFINE COM CLASS(#PRIM PHBN) NAME(#CLOSEBTN) CAPTION('C DEFINE\_COM CLASS(#PRIM\_LTVW) NAME(#LTVW\_2) COLUMNHEA DEFINE\_COM CLASS(#PRIM\_LVCL) NAME(#LVCL\_9) DISPLAYPOSIT DEFINE COM CLASS(#STD\_NUM) NAME(#I) DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#SHOWHIDEBTN) CAPTI( DEFINE\_COM CLASS(#PRIM\_LABL) NAME(#LABL\_1) CAPTION('The c DEFINE\_COM CLASS(#PRIM\_STBR) NAME(#STBR\_1) DISPLAYPOSITI DEFINE\_COM CLASS(#PRIM\_LABL) NAME(#LABL\_2) CAPTION('Dout click an employee to insert data.') DISPLAYPOSITION(2) ENABLED(False) DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#EXITBTN) CAPTION('Exi

Def\_list name(#skills) fields(#skilcode #SkilDesc #dateacqr #grade #comment

EVTROUTINE HANDLING(#STARTBTN.Click)

\*Start Word by setting a reference to it. Word has been enrolled as ActiveX co SET\_REF COM(#WordApp) TO(\*CREATE\_AS #VA\_WORD.application) \*

\*If Word has been started, enable some controls and issue a message if\_ref com(#wordapp) is\_not(\*null)

set com(#showhidebtn #gpbx\_1 #rdbn\_1 #rdbn\_2 #rdbn\_3 #Addbtn #labl\_1 #e
execute infmessage 'Word has been started.'

endif

ENDROUTINE

EVTROUTINE HANDLING(#showhidebtn.Click) \*If Word is hidden make it visible and vice versa if cond('#wordapp.visible \*eq True') SET #WordApp Visible(false) else SET #WordApp Visible(true) endif ENDROUTINE

EVTROUTINE HANDLING(#RDBN\_1.Click) \*Set the size of the Word window using the Windowstate property of the Word IF COND('#rdbn\_1.buttonchecked \*eq true') SET #WordApp Windowstate(VA\_WORD.wdWindowStateMinimize) endif ENDROUTINE EVTROUTINE HANDLING(#RDBN\_2.Click) if cond('#rDbn\_2.buttonchecked \*eq True') SET #WordApp Windowstate(VA\_WORD.wdWindowStateNormal) endif ENDROUTINE EVTROUTINE HANDLING(#RDBN\_3.Click) if cond('#rdbn\_3.buttonchecked \*eq True') SET #WordApp Windowstate(VA\_WORD.wdWindowStateMaximize) endif ENDROUTINE

EVTROUTINE HANDLING(#EXITBTN.Click)

\* Close Word without saving any changes

```
invoke method(#WordApp.Quit) savechanges(0)
*
```

\*Disable controls in the Word Application group box except for the Start Word set com(#showhidebtn #gpbx\_1 #rdbn\_1 #rdbn\_2 #rdbn\_3 #Addbtn #labl\_1 #e \*

\*Disable controls in the Data for New Document group box

set com( #gpbx\_3 #labl\_2) enabled(false)

clr\_list #ltvw\_1

\*Clear lists and disable controls in the Work with Document group box clr\_list #ltvw\_2

```
set com(#gpbx_4 #savebtn #printbtn #closebtn) enabled(false)
execute infmessage 'Word has been closed.'
```

ENDROUTINE

#### EVTROUTINE HANDLING(#ADDBTN.Click)

\*Add a new document and base it on the vl\_sam99.dot Word template USE BUILTIN(TCONCAT) WITH\_ARGS(\*PART\_DIR\_SOURCE VL\_SAM invoke method(#WordApp.documents.add) template(#STD\_QSEL) add\_retval \*

\*Add employee information to #ltvW\_1 select fields(\*all) from\_file(pslmst) add\_entry #ltvw\_1 endselect

\*Enable and show controls in the Data for New Document group box set com(#gpbx\_3 #labl\_2) enabled(true)

execute infmessage 'A new document has been opened.'

\*Update the list of documents execute doclist

\*disable the Add Document button until the employee information has been in: set #addbtn enabled(false) ENDROUTINE

\*Add employee information to the document EVTROUTINE HANDLING(#ltvw\_1.DoubleClick) \*Concatenate first name and surname to get #Fullname Use BConcat (#GiveName #SurName) (#FullName) \*Locate the Word bookmark 'fullname' in the document invoke method(#WordDoc.Bookmarks.item<'fullname'>.select) \*Insert the value of the #fullname field invoke method(#WordApp.Selection.TypeText) text(#fullname)

invoke method(#WordDoc.Bookmarks.item<'address1'>.select)
invoke method(#WordApp.Selection.TypeText) text(#address1)

invoke method(#WordDoc.Bookmarks.item<'address2'>.select)
invoke method(#WordApp.Selection.TypeText) text(#address2)

invoke method(#WordDoc.Bookmarks.item<'address3'>.select)
invoke method(#WordApp.Selection.TypeText) text(#address3)

invoke method(#WordDoc.Bookmarks.item<'postcode'>.select)
use numeric\_string #postcode #std\_texts
invoke method(#WordApp.Selection.TypeText) text(#std\_texts)

invoke method(#WordDoc.Bookmarks.item<'phonehme'>.select)
invoke method(#WordApp.Selection.TypeText) text(#phonehme)

invoke method(#WordDoc.Bookmarks.item<'phonebus'>.select)
invoke method(#WordApp.Selection.TypeText) text(#phonebus)

invoke method(#WordDoc.Bookmarks.item<'writernam'>.select)
invoke method(#WordApp.Selection.TypeText) text(#WordApp.UserName)

invoke method(#WordDoc.Bookmarks.item<'homepage'>.select)
invoke method(#WordApp.Selection.TypeText) text(www.lansa.com)

invoke method(#WordDoc.Bookmarks.item<'fulldate'>.select)
invoke method(#WordApp.Selection.TypeText) text(#datec)

\*Get the skills information for the employee and add it to the working list #skil Select \*all From\_File(PslSkl) with\_Key(#Empno) Fetch #SkilDesc from\_File(SklTab) With\_Key(#SkilCode) Keep\_Last(50) Add\_Entry #Skills EndSelect

invoke method(#WordDoc.Bookmarks.item<'skills'>.select)
Selectlist #skills
\*Insert the value of the #skilcode field
invoke method(#WordApp.Selection.TypeText) text(#skilcode)
\*Move cursor right to the next table cell
invoke method(#wordapp.Selection.MoveRight) Unit(1) Count(1)

invoke method(#WordApp.Selection.TypeText) text(#skildesc)
invoke method(#wordapp.Selection.MoveRight) Unit(1) Count(1)

invoke method(#WordApp.Selection.TypeText) text(#grade)
invoke method(#wordapp.Selection.MoveRight) Unit(1) Count(1)

invoke method(#WordApp.Selection.TypeText) text(#comment)
invoke method(#wordapp.Selection.MoveRight) Unit(1) Count(1)

\*Add a new row to the table invoke method(#wordapp.Selection.InsertRows) endselect \*Delete the last empty row of the table invoke method(#wordapp.selection.Rows.Delete)

\*Enable and disable components set com(#gpbx\_4 #ltvw\_2 #savebtn #closebtn #printbtn) enabled(true) set com(#gpbx\_3 #labl\_2) enabled(false) clr\_list #ltvw\_1 set #addbtn enabled(true)

execute infmessage 'Employee details have been added to document.' ENDROUTINE

EVTROUTINE HANDLING(#LTVW\_2.ItemGotFocus) \*activate the selected document invoke method(#wordapp.documents.item<#std\_text>.activate)

\*set #ltvw\_2.currentitem selected(true) ENDROUTINE

EVTROUTINE HANDLING(#PRINTBTN.Click) \*Print out the document. invoke method(#WordDoc.PrintOut)

\*Alternative methods:

\* invoke method(#WordApp.Dialogs.item<VA\_WORD.wdDialogFilePrint>.sh \* invoke method(#WordApp.ActiveDocument.PrintOut) execute infmessage 'Document has been sent to the printer.' ENDROUTINE

EVTROUTINE HANDLING(#SAVEBTN.Click) \*Open the Save dialog invoke method(#WordApp.Dialogs.item<VA\_WORD.wdDialogFileSaveAs>.s

\*update document list execute doclist execute infmessage 'Document has been saved.' ENDROUTINE

EVTROUTINE HANDLING(#CLOSEBTN.Click) \*if a document is open, close it without saving any changes if cond('#WordApp.Documents.Count \*gt 0') invoke method(#wordapp.activedocument.close) savechanges(0) endif execute doclist execute infmessage 'Document has been closed.' ENDROUTINE

SUBROUTINE NAME(DOCLIST) \*get the number of open Word documents change #std\_num #WordApp.Documents.Count \*Clear #ltvw\_2 and then add the names of all open documents clr\_list #ltvw\_2 begin\_loop using(#i) FROM(1) TO(#std\_num) change #std\_text #wordapp.documents.item<#i.value>.name\_COM add\_entry #ltvw\_2 end\_loop

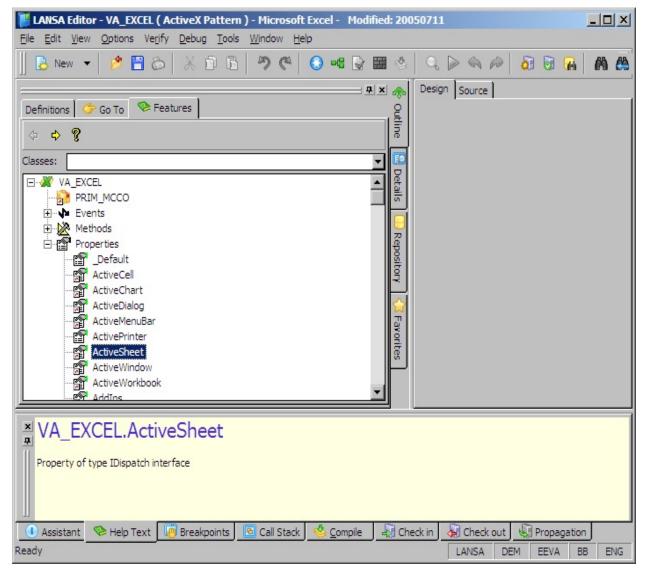
if cond('#std\_num \*gt 0')
\*Set a reference to the currently active document into the variable #WordDoc
SET\_REF COM(#WordDoc) TO(#WordApp.ActiveDocument)
endif

endroutine

\* Information Message created by template VL\_BBSTSBR SUBROUTINE INFMESSAGE ((#EX\_TEXT \*RECEIVED)) define #ex\_text \*char 132 message msgid(dcm9993) msgf(dc@m01) msgdta(#ex\_text \*blanks) ENDROUTINE END\_COM

## 7.2.5 Properties of Type IDispatch

Depending on how the type library of an ActiveX component is set up, a property can be returned to Visual LANSA as type IDispatch instead of a specific type. With the IDispatch interface the properties, events and methods of the object will only be resolved at run-time. For example the Worksheet object in Excel is of type IDispatch.



Because the type of the object is not known, statements using its properties, events and methods will return an error "Component IDispatch does not contain property/method/event X". For example this statement will cause the error:

invoke method(#va\_excel.worksheets.item<'blad2'>.activate)

So, in order to use an object of type IDispatch interface, you must first define the object with a dynamic reference and then explicitly set the reference to it. Only then can you use the object's properties, methods and events.

For example to activate an Excel worksheet, you need to first define it:

Define\_Com Class(#VA\_EXCEL.Worksheet) Name(#WorkSheet) Reference(\*

Then in your code set the reference to it:

Set\_Ref Com(#Worksheet) To(\*Dynamic #Excel.Worksheets.Item<'blad2'>)

And only then use the Activate method: Invoke Method(#Worksheet.Activate)

The feature help (F2) will show you that the type of the object is IDispatch interface. To find out the actual type of the object, refer to whatever documentation is provided with the ActiveX control.

↑ 7.2 Integrating with ActiveX-Enabled Applications

## 7.3 ActiveX Wrapping

You can expose LANSA components (forms and reusable parts) as industry standard ActiveX controls on Microsoft Windows. In this way you can expose the functionality of your components to a range of third party applications.

Using ActiveX wrapping you can:

- Integrate LANSA applications with other tools that support ActiveX (e.g. Borland's Delphi and Microsoft's Visual Studio).
- Make LANSA business logic more accessible to other tools
- Enable Independent Software Vendors (ISV) to expose application business logic to their customers.

Examples in this guide have been provided in Visual Basic 6.0 due to its common use in the market place. Tests have also been performed using Borland's Delphi and Visual Studio.NET (C## and Visual Basic). For specific syntax for these tools, please refer to the user documentation of the respective tool.

Before trying out the examples, please check that the 7.3.1 Prerequisites are satisfied.

- 7.3.1 Prerequisites
- 7.3.2 Creating an ActiveX Component
- 7.3.3 Examples
- 7.3.4 Troubleshooting
- 7.3.5 Syntax differences from Visual Basic 6.0 to Visual Basic.NET
- 7.3.6 Source Code
- ↑ 7. ActiveX Controls

## 7.3.1 Prerequisites

To complete the examples you need:

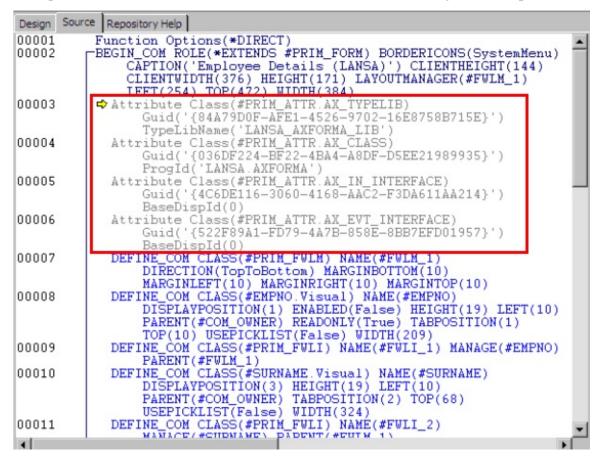
- Basic knowledge of Visual LANSA
- Basic knowledge of Visual Basic
- Visual LANSA development and runtime environments at V11.0 or above.
- Visual Basic development and runtime environments with Service Pack 5 (on the same machine as Visual LANSA)
- SET material and LANSA Personnel System application data installed in the repository

## 7.3.2 Creating an ActiveX Component

1. To expose a LANSA form or reusable part as an ActiveX control, open it in the editor and choose the Set ActiveX Attributes option in the ribbon:



The required ActiveX attributes are added to the source of your component:



Compile the component to create the ActiveX control.

## 7.3.3 Examples

Following are some examples to show how to wrap LANSA components as ActiveX components. Before you can start using these examples, first complete the General Setup Tasks.

The examples provided are:

Case 1 - Show a Form

This example shows you the essentials of exposing properties, events and methods.

Case 2 - Embed a Reusable Part

This example shows you how to embed a Visual LANSA Re-usable part inside a Visual Basic form.

Case 3 - Use Components as Business Objects

This example shows you how to:

- Use non-visual re-usable parts
- Dynamically build and manage objects using collections
- Invoke business rules from the LANSA repository
- Handle messages issued from LANSA

## **General Setup Tasks**

Session initialization Add the Common Controls to the Project Add a Reference to the LANSA Session Object

## Session initialization

At runtime, Visual Basic needs to know which LANSA environment to connect to, such as the partition and database. Specify this information in a session configuration file. This file uses the same parameters as X\_RUN.

To create a session configuration file:

- 1. Open Notepad
- 2. Copy the sample file below.
- 3. Modify the parameters to suit your site requirements.

These parameters can also be set programmatically with or without a session configuration file.

4. Save the file to the directory that you will use to create your Visual Basic samples and exit Notepad.

## **Example Session.cfg files**

LANG=e.g. ENG

PART=partition e.g SET

DBII=your database's name

DBUS=database user - e.g. DBA

PSWD=database user password - e.g. SQL

GUSR=\*NONE

HLPC=4392492

LOCK=YES

TASK=task\_id

DATF=DMY

DBUG=N

ITRO=N

INIT=

TERM=

DBIT=database type - e.g SQLANYWHERE,...

📕 session.cfg - Notepad	
<u>File Edit Format View Help</u>	
LANG=ENG	A
PART=DEM	
DBII=LXLANSA1	
DBUS=DBA	
PSWD=SQL	
GUSR=*NONE	
HLPC=4392492	
LOCK=YES	
TASK=BB	
DATF=DMY	
DBUG=N	
ITRO=N	
INIT=	
TERM=	
DBIT=SQLANYWHERE or DBIT=MSSQLS	
	-
4	

↑ General Setup Tasks

## Add the Common Controls to the Project

The Windows Common Controls need to be included in your VB project.

1. Open your Visual Basic development environment and create a new project using the standard Visual Basic project template.



2. Add the common controls to the toolbox. Right click in the toolbox and, from the pop-up menu, select Components:



3. Select Microsoft Windows Common Controls 6.0:

Microsoft Works Calendar 6.0 Type Library	Controls Designers Insertable Objects		
Microsoft Winsock Control 6.0         Microsoft Works Calendar 6.0 Type Library         ▲<	Microsoft Shell Controls And Automation     Microsoft SysInfo Control 6.0     Microsoft Tabbed Dialog Control 6.0 (SP3)     Microsoft Tablet PC Type Library, version 1.0     Microsoft Visio 11.0 Drawing Control Type Librar     Microsoft Windows Common Controls 5.0 (SP2)     Microsoft Windows Common Controls 2 5.0 (SP2)     Microsoft Windows Common Controls 2 5.0 (SP2)     Microsoft Windows Common Controls 2 5.0 (SP2)		
	Microsoft Winsock Control 6.0	Browse	
Location: C:\WINDOWS\system32\MSCOMCTL.OCX	Microsoft Windows Common Controls 6.0 (SP6)	)CX	

3. Click OK. The Common Controls are now available in the toolbox:

					×
		Gene	eral		
k		A	abl	ו-	
	◄	¢		<u>ا</u>	
٩F	1	Ö			
Ē	6	/			
OLE					
	999 99	đ	₽		

↑ General Setup Tasks

## Add a Reference to the LANSA Session Object

You need to add a reference to the LANSA ActiveX library in order to use LANSA ActiveX components.

- 1. Select References from the Project menu to add a reference to the LANSA ActiveX library.
- 2. Select the box LANSA ACTIVEX LIB 1.0.

References - Project1.vbp	×
<u>A</u> vailable References:	ОК
<ul> <li>✓ Visual Basic For Applications</li> <li>✓ Visual Basic runtime objects and procedures</li> </ul>	Cancel
Visual Basic objects and procedures OLE Automation	Browse
✓ LANSA ACTIVEX LIB 1.0      ✓ Microsoft Data Formatting Object Library 6.0 (SP4)	
LANSA PRIM LIB 1.0     IAS Helper COM Component 1.0 Type Library     IAS RADIUS Protocol 1.0 Type Library	y <u>H</u> elp
8021XConfig 1.0 Type Library     Acrobat Access 2.0 Type Library	
AcroIEHelper 1.0 Type Library	
Active Setup Control Library	
LANSA ACTIVEX LIB 1.0	
Location: C:\Program Files\LANSAIndependent11\X_WIN	95\X_LANSA\E:
Language: Standard	

- 3. Click OK.
- 4. Save the project and form into a subfolder under the folder in which you saved your Session Configuration file. Use the default names of Project<n>.vbp and Form<n>.frm.

The setup is now complete.

↑ General Setup Tasks

## Case 1 - Show a Form

This example shows the essentials of how to expose properties, events and methods of an ActiveX LANSA form:

🖨 Case 1 - VB		-OX		
Demonstrates the events and metho	e essentials of how to expo ods	se properties,		
Employee Number:	A1012			
Employee Details				
Name:	PATRICK	Show		
	PAUL	Update		
Salary:	26456.04	Liose Visual LANSA		
		Employee Details (LA	NSA)	×
		Employee Number	A1012	
		Employee Given Name(s)	PATRICK	
		Employee Surname	PAUL	
		Employee Salary	26,456.04	

Source that makes up this example:

Name	Туре	Description
AXFORMA	Visual LANSA Component	Visual LANSA Source
Project1	Visual Basic Project	Project Definition
Form1	Visual Basic Form	Visual Basic - Code
Session.cfg	Configuration file	Session Configuration (Session.cfg)

## Add the Visual LANSA Component

In this step you create an ActiveX LANSA form to display employee details:

Employee Details (LAN	ISA)	×
Employee Number	A1012	
Employee Given Name(s)	PATRICK	
Employee Surname	PAUL	
Employee Salary	26,456.04	

To create the ActiveX component:

- 1. Open the LANSA editor and create a new component named AXFORMA, type Form.
- 2. Copy and paste the source code from Case 1 Visual LANSA Source into the form.
- 3. Select the Set ActiveX Attributes command in the ribbon. (The sample code you have copied already has these attributes, but setting them again does not do any harm.)

This step exposes the Visual LANSA component as an ActiveX Control. You can see what the command does by first deleting the existing four ActiveX attribute statements (Attribute Class(#PRIM\_ATTR.AX\_) from the source before selecting the menu option)

Compile the form.

1 Case 1 - Show a Form

## Add the Visual Basic Form

In this step you create a Visual Basic form which uses the LANSA ActiveX form:

🕞, Case 1 - VB		
Demonstrate events and m	s the essentials of how to ethods	o expose properties,
Employee Num	ber: A1012	
Employee Detail	s	
Name:	PATRICK	[Show]
	PAUL	Update
Salary:	26456.04	Close Visual LANSA

To create this form:

- 1. Close Visual Basic.
- 2. Start Notepad, open the form (Form<n>.frm) created in the General Setup Tasks.
- 3. Replace the code in Notepad with the source code from Case 1 Visual Basic Object, then Save.
- 4. Start Visual Basic and open the Project (Project<n>.vbp) created in the General Setup Tasks.
- 5. Open the form (Form<n>.frm)
- 6. Copy the source code from Case 1 Visual Basic Code to the form's code, then Save.
- 7. Right click in the Visual Basic toolbox and select components from the popup menu.
- 8. Scroll down and check LANSA.AXFORMA. This will add a LANSA icon to the toolbox.



10. Click on the AXFORMA icon to select it and on the form drag out a rectangle and release the left mouse key. The LANSA ActiveX form has now been added to the Visual Basic form:

ፍ Case 1 - VB		
Demonstrates events and m	the essentials of how to ex ethods	pose properties,
Employee Numl		
Employee Details		
Name:	Give Name	Show
	Surname	Update
Salary:	Salary	Llose Visual

11. Display the properties of the form and change its name from AXFORMA1 to AXFORMA.

#### Notes:

• If your Toolbox is not displayed, click View then Toolbox.

- To view the form's user interface design, Click View, then click Object from the Menu or the Project Pane.
- 12. In the Visual Basic LOAD routine (Private Sub Form\_Load) modify the login script to your requirements. For example:

Call ConnectToLansa("user1", "mypassword", "c:\ActiveXTests\session.cf{

13. Save, then compile the Visual Basic Project using the Make Project<n>.exe option in the File menu.

You can now view the sample Personnel data by executing the program and typing in an Employee Number, then clicking on Show.

↑ Case 1 - Show a Form

## **Case 2 - Embed a Reusable Part**

This example shows how you can embed a LANSA ActiveX reusable part to a Visual Basic form. The LANSA ActiveX component is the grid showing skill details for an employee:

nployee Numb	JA	1012 Show Skills	
Skill	Grade	Skill Full Description	
ADMIN1	D	Administratn Part 1	
ADMIN2	P	Administratn Part 2	
CS	P	Computer Science Deg	
HIS	Р	History Degree	
INTRO	Р	Company Introduction	
KEY	P	Keyboard Skills	
MANAGE1	D	Management Course 1	
MANAGE3	F	Management Course 3	
MARKET1	D	Marketing Course 1	
OTH	D	Other Degree Course	
	8		

Source that makes up this example:

Name	Туре	Description
AXPANELA	Visual LANSA Component	Visual LANSA Source
Project1	Visual Basic Project	Project Definition
Form1	Visual Basic Form	Visual Basic - Code
Session.cfg	Configuration file	Session Configuration (Session.cfg)

## Add the Visual LANSA Component

In this step you create the LANSA ActiveX reusable part containing a grid:

	Imbedded version of ActiveX Interaction 101       Imployee Number:         A1012       Show Skills		
	Skill	Grade	Skill Full Description
	ADMIN1	D	Administratn Part 1
	ADMIN2	Р	Administratn Part 2
	CS	Р	Computer Science Deg
	HIS	Р	History Degree
	INTRO	Р	Company Introduction
	KEY	Р	Keyboard Skills
	MANAGE1	D	Management Course 1
	MANAGE3	F	Management Course 3
	MARKET1	D	Marketing Course 1
	OTH	D	Other Degree Course
Em	ployee skills s	elected	

To create the LANSA ActiveX control:

- 1. Open the LANSA editor and create a new component named AXPANELA, type Reusable Part.
- 2. Copy and paste into the Visual LANSA Editor the source code from Case 2 Visual LANSA Source.
- 3. Use the Set ActiveX Attributes command in the ribbon.
- 4. Compile the component.
- ↑ Case 2 Embed a Reusable Part

## Add the Visual Basic Form

In this step you create the Visual Basic form which contains the LANSA ActiveX control:

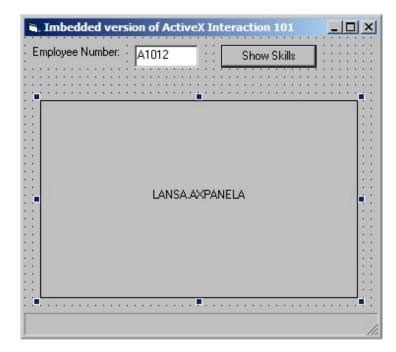
Imbedded version of ActiveX Interaction 101       Employee Number:       A1012   Show Skills							
F	Skill	Grade	Skill Full Description				
	ADMIN1	D	Administratn Part 1				
8	ADMIN2	Р	Administratn Part 2				
8	CS	Р	Computer Science Deg				
	HIS	Р	History Degree				
	INTRO	Р	Company Introduction				
	KEY	Р	Keyboard Skills				
8	MANAGE1	D	Management Course 1				
	MANAGE3	F	Management Course 3				
	MARKET1	D	Marketing Course 1				
	OTH	D	Other Degree Course				
Employee skills selected							

To create the form:

- 1. Complete the General Setup Tasks to create a new Visual Basic project.
- 2. Close Visual Basic.
- 3. Start Notepad, open the form (Form<n>.frm) created in the General Setup Tasks.
- 4. Replace the code in Notepad with the source code from Case 2 Visual Basic Object, then save it.
- 5. Open the Project (Project<n>.vbp) in your Visual Basic development environment.
- 6. Open the form (Form<n>.frm)
- 7. Replace the code with the source code from Case 2 Visual Basic Code, then Save it.
- 8. Right click in the Visual Basic toolbox and select Components from the popup menu.

Controls       Designers       Insertable Objects         IStudio Active Designer Controls       ISWizard 1.0 Type Library         LANDesk Date/Time controls       ISWizard 1.0 Type Library         LANSA ACTIVEX LIB 1.0       Image: See See See See See See See See See S	Components		x
ISWizard 1.0 Type Library         LANDesk Date/Time controls         LANSA ACTIVEX LIB 1.0         LANSA.AXFORMA         LANSA.AXFORMA         LayoutDTC 1.0 Type Library         LDVPCtls ActiveX Control module         LDVPDIgs ActiveX Control module         LDVPUI ActiveX Control module         LDVPView LANDesk Virus Protect Task module         LDVPView LANDesk Virus Protect Task module         LDVPNiew LANDesk Virus Protect Task module	Controls Designers Insertable Objects		
LayoutDTC 1.0 Type Library LDVPCtls ActiveX Control module LDVPDlgs ActiveX Control module LDVPTaskFS OLE Control module LDVPUI ActiveX Control module LDVPUI ActiveX Control module Enowse Selected Items Only LANSA.AXPANELA	ISWizard 1.0 Type Library LANDesk Date/Time controls LANSA ACTIVEX LIB 1.0 LANSA.AXFORMA LANSA.AXFORMA		
LANSA.AXPANELA	LayoutDTC 1.0 Type Library LDVPCtls ActiveX Control module LDVPDlgs ActiveX Control module LDVPTaskFS OLE Control module LDVPUI ActiveX Control module	<u>B</u> rowse	
OK Cancel Apply	Location: c:\\x_l\x_dem\execute\axpanela.dll		

- 9. Scroll down and check LANSA.AXPANELA. This will add an icon for the AXPANEL control to the toolbox
- 10. Click on the AXPANELIA icon to select it and on the form drag out a rectangle and release the left mouse key to include the LANSA ActiveX control to the form:



11. View the properties of the ActiveX control and change the name from AXPANELA1 to AXPANELA.

Notes:

- If your Toolbox is not displayed, click View then Toolbox.
- To view the forms user interface design, Click View, then click Object from the Menu or by using the Project Pane.
- 12. In the LOAD routine modify the login script below to your requirements. For example:

Call ConnectToLansa("user1", "mypassword", "c:\ActiveXTests\session.cf{

13. Save, then compile the Visual Basic Project using the Make Project<n>.exe option in the File menu.

You can now view the sample Personnel data by executing the program and typing in an Employee Number, then clicking on Show.

1 Case 2 - Embed a Reusable Part

## **Case 3 - Use Components as Business Objects**

In this example you create and use two non-visual LANSA ActiveX components (business objects) which provide functionality to manage employee and skills information in the LANSA Personnel System demonstration files:

<b>i</b> , 0	ase 3 - VB		
	Department:	ADM	Search
	Surname	Department	Given N. 🔺
	BLAKE BLOGGS BROWN	MIS FLT ADM	JOHN FRED JC FREDD'
	BROWN DICKENS	INF LEG	VERONI CHARLE
	DOUGLAS	ADM	ADAM P
	JACKSON	SD	STEPHE
	JONES LEWIS	ADM GAC	BEN TONY
	LINCOLN	INF	PAUL
	MAXWELL	GAC	ANDRE\
	Ba Employee Detail Number: Name: Salary:		ard
		Update	
Г			-

This example shows you how to:

- Dynamically build and manage objects using collections
- Invoke business rules from the LANSA repository
- Handle messages issued from LANSA

Source that makes up this example:

Name Type

#### Description

AXOBJECTA	Visual LANSA Component	Reusable Part AXOBJECTA
AXOBJECTB	Visual LANSA Component	Reusable Part AXOBJECTB
Project1	Visual Basic Project	Project Definition
Form1	Visual Basic Form	Visual Basic - Code
Session.cfg	Configuration file	Session Configuration (Session.cfg)

## Add the Visual LANSA Components

In this step you create two non-visual ActiveX LANSA business objects which provide functionality for managing employee information. These objects are used by the Visual Basic form.

- 1. Open the LANSA editor and create a new component named AXOBJECTA, type Reusable Part.
- 2. Copy and paste into the Visual LANSA Editor the source code for Reusable Part AXOBJECTA into the component.
- Use the Set ActiveX Attributes command in the ribbon.
   This step exposes the Visual LANSA component as an ActiveX Control.
- 5. Repeat steps 1 to 3 for component Reusable Part AXOBJECTB.
- 6. Compile components AXOBJECTB and AXOBJECTA.
- ↑ Case 3 Use Components as Business Objects

## Add the Visual Basic Form

In this step you create the Visual Basic form which uses the non-visual LANSA ActiveX components to manage employee information in the LANSA Personnel System demonstration files:

- 1. Complete the General Setup Tasks to create a new Visual Basic project.
- 2. Close Visual Basic.
- 3. Start Notepad and open the form (Form<n>.frm) created in the General Setup Tasks.
- 4. Replace the code in Notepad the source code from Case 3 Visual Basic Object, then Save.
- 5. Open the Project (Project<n>.vbp) in your Visual Basic development environment.
- 6. Open the form (Form<n>.frm)
- 7. Replace the code with the source code from Case 3 Visual Basic Code, then Save.
- 8. Select the Project menu and then the References...option.
- 9. Scroll down and select LANSA.AXOBJECTA and LANSA.AXOBJECTB. Click OK. This will add a Reference for each object to the project.

#### Notes:

- If your Toolbox is not displayed, click View then Toolbox.
- To view the form's user interface, Click View, then click Object from the Menu or by using the Project Pane.
- If you have an empty list at runtime then check the view style, it should be lvwReport
- You do not need to add the LANSA ActiveX business objects to the toolbox or the form.

## 10. In the LOAD routine modify the login script below to your requirements. For example:

Call ConnectToLansa("user1", "mypassword", "c:\ActiveXTests\session.cf

11. Save, then compile the Visual Basic Project using the Make Project<n>.exe option in the File menu.

1 Case 3 - Use Components as Business Objects

## 7.3.4 Troubleshooting

Persistent Visual Basic Crashes Unable to compile LANSA component due to lock Visual Basic forms are blank after cut/paste examples Error: Method 'ConfigFile' of object 'ISession' failed Making an executable causes application to fail when run in Visual Basic 6.0 Events do not fire

## **Persistent Visual Basic Crashes**

For no specific reason the Visual Basic development environment crashes. Install Service Pack 5.

## Unable to compile LANSA component due to lock

You may receive a "LINK: fatal error LNK1104...." on the final stages of a Visual LANSA component compile. This is caused by the Visual Basic development environment locking components if they are referenced e.g. if they are added to its component toolbar. Close Visual Basic and re-compile Visual LANSA component.

## Visual Basic forms are blank after cut/paste examples

As Visual Basic form layouts are not available in the source you must either:

1. Use Notepad to cut/paste the supplied object source into a Visual Basic form prior to viewing them

or

2. Add the controls manually using the screen captures included in the provided examples as a basis.

## Error: Method 'ConfigFile' of object 'ISession' failed

Either your Session initialization file (session.cfg) is not valid or can't be found. Check the parameters are correct and that the file session.cfg contains the correct information.

# Making an executable causes application to fail when run in Visual Basic 6.0

After Making an executable, running the application from the Visual Basic 6.0 environment behaves oddly. Calls to LANSA ActiveX components may fail. If

there are multiple embedded Visual LANSA Panels, they may all look the same. This is caused by some incorrectly cached information in Visual Basic 6.0. This may improve on the second invocation, and will be fixed by quitting Visual Basic 6.0 and restarting.

## **Events do not fire**

Visual Basic 6.0 does not like an event with an underscore ( '\_' ) in the name, this may make all events fail! In Visual LANSA, change the 'Name' property in the event's ActiveX attribute and recompile eg:

Define\_Evt Name(OK\_Button\_Pressed) Attribute Class(#PRIM\_ATTR.AX\_EVT\_MEMBER) Dispid(9) Name('OI

# 7.3.5 Syntax differences from Visual Basic 6.0 to Visual Basic.NET

Some tips for using .NET syntax: Add References to the LANSA Objects Initialize the Session Create a Visual LANSA Component Imbed a Visual LANSA Component on a Visual Basic.NET Form ↑ 7.3.3 Examples

## Add References to the LANSA Objects

- 1. Click *Project*, and then *Add References*.
- 2. Select the COM tab, and then scroll down to the LANSA Components.
- 3. Select LANSA ACTIVEX LIB 1.0 and LANSA.<Component Name>.

```
' Add a global reference to a LANSA Session object interface
Private Session As LANSA_ACTIVEX_LIB.Isession
```

'Add a global reference to a VL Component interface

Private MyComponent As LANSA\_MYCOMP\_LIB.I\_MYCOMP

' Add code to initialize the session object and connect to LANSA.

```
Session = New LANSA_ACTIVEX_LIB.Session()
Session.ConfigFile = "./Session.cfg"
Session.Connect
```

' Ask the Session to create the VL component.

MyComponent = Session.CreateComponent("MYCOMP")

Access the component's properties and methods as per any ActiveX control. The application will disconnect from LANSA when the Session object is destroyed.

↑ 7.3.5 Syntax differences from Visual Basic 6.0 to Visual Basic.NET

#### **Initialize the Session**

Private Session As LANSA\_ACTIVEX\_LIB.ISession Private Sub ConnectToLansa( \_ ByVal username As String, \_ ByVal password As String)) Try

' Create the Session object Session = New LANSA\_ACTIVEX\_LIB.Session

' Set the session configuration file Session.ConfigFile = "./Session.cfg"

' Set the User and Password Session.SetConnectParam("USER", username) Session.SetConnectParam("PSPW", password)

' Connect to LANSA Session.Connect

Catch ex As Exception MessageBox.Show("Exception : " + ex.Message) End Try

End Sub

↑ 7.3.5 Syntax differences from Visual Basic 6.0 to Visual Basic.NET

## **Create a Visual LANSA Component**

Use Session.CreateComponent or .AddComponent in Visual Basic.NET.

Private MyComponent As LANSA\_MYCOMP\_LIB.MYCOMP

If Session.IsConnected then MyComponent = Session.CreateComponent("MYCOMP") End if

↑ 7.3.5 Syntax differences from Visual Basic 6.0 to Visual Basic.NET

## Imbed a Visual LANSA Component on a Visual Basic.NET Form

To embed a VL Panel Reusable Part on a Visual Basic form, or respond to the control's events, it is necessary to place the component using Drag and Drop from the toolbar:

- 1. View the Form
- 2. Select the Components section in the Toolbox
- 3. Right-Click on the Toolbox, select Customize Toolbox...
- 4. Select the *COM Components* tab (default)
- 5. Scroll down to **LANSA.MYCOMP** and tick the box.
- 6. Press OK, a small LANSA icon should appear on the toolbar.
- 7. Drag the icon onto the form, drag out a rectangle and release.
- 8. If the component is a Panel, a blank rectangle of the Panel's size should appear with the name "LANSA.MYCOMP".
- 9. If the component is an Object or a Form, it will appear as an icon at design time.
- 10. A component will have been automatically defined and the creation code added in your Visual Basic form after the drag drop. The default name will be **axMYCOMP1**.
- 11. It is now necessary to add this component to the Session. Don't call "Session.CreateComponent", call "Session.AddComponent".

Session.AddComponent(axMYCOMP1.GetOcx())

- 12. A Panel component should now appear "live" when the application is run.
- 13. An Object will not appear, a Form will not appear until ShowForm is called. The component's methods, properties AND events are now accessible to Visual Basic.
- ↑ 7.3.5 Syntax differences from Visual Basic 6.0 to Visual Basic.NET

### 7.3.6 Source Code

Create and insert your session configuration file (Session.cfg) as described in General Setup Tasks.

Case 1 - Show a Form

Case 1 - Show a Form

Case 2 - Embed a Reusable Part

Case 3 - Use Components as Business Objects

# Case 1 - Show a Form

🖷, Case 1 - VB		
Demonstrates t events and met		v to expose properties,
Employee Numbe	r: A1012	
Employee Details-		
Name:	Give Name	[Show]
	Surname	Update
Salary:	Salary	Close VL

For code samples, go to:

- Case 1 Visual LANSA Source
- Case 1 Visual Basic Object
- Case 1 Visual Basic Code

### **Case 1 - Visual LANSA Source**

```
Function Options(*DIRECT)
```

BEGIN\_COM ROLE(\*EXTENDS #PRIM\_FORM) BORDERICONS(System Attribute Class(#PRIM ATTR.AX TYPELIB) Guid('{385B3936-74A3-4700-AA82-D7D9BEE8EA46}') TypeLibName('LANSA AXFORMA LIB') Attribute Class(#PRIM ATTR.AX CLASS) Guid('{BB00E891-AA66-4DAA-ACC8-7B026B934A78}') ProgId('LANSA.AXFORMA') Attribute Class(#PRIM ATTR.AX IN INTERFACE) Guid('{503CD2F3-340E-48B0-867B-D10667D1E57A}') BaseDispId(0) Attribute Class(#PRIM ATTR.AX EVT INTERFACE) Guid('{22D638D0-B9E0-4FAE-99E3-EA39F884EC97}') BaseDispId(0) DEFINE COM CLASS(#PRIM FWLM) NAME(#FWLM 1) DIRECTION(7 DEFINE COM CLASS(#EMPNO.Visual) NAME(#EMPNO) DISPLAYPOSI DEFINE COM CLASS(#PRIM FWLI) NAME(#FWLI 1) MANAGE(#EMP DEFINE COM CLASS(#SURNAME.Visual) NAME(#SURNAME) DISPLA DEFINE COM CLASS(#PRIM FWLI) NAME(#FWLI 2) MANAGE(#SUR DEFINE COM CLASS(#GIVENAME.Visual) NAME(#GIVENAME) DISPL DEFINE COM CLASS(#PRIM FWLI) NAME(#FWLI 3) MANAGE(#GIVI DEFINE COM CLASS(#SALARY.Visual) NAME(#SALARY) DISPLAYPO DEFINE COM CLASS(#PRIM FWLI) NAME(#FWLI 4) MANAGE(#SAL

Define\_Evt Name(uEmployeeFound)

Attribute Class(#PRIM\_ATTR.AX\_EVT\_MEMBER) Dispid(0) Name('uEmpl Define\_Map For(\*input) Class(#io\$sts) Name(#uResult)

Define\_Pty Name(uEmployeeGiveName) Get(\*auto #givename) Set(\*auto #gi Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(0) Name('uEmploy Define\_Pty Name(uEmployeeSurname) Get(\*auto #surname)

Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(1) Name('uEmploy Define\_Pty Name(uEmployeeSalary) Get(\*auto #salary) Set(\*auto #salary) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(2) Name('uEmploy \*

Mthroutine Name(uShowEmployee)

Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(3) Name('uShowEı Define\_Map For(\*input) Class(#empno) Name(#i\_empno)

Change Field(#EMPNO) To(#I\_EMPNO)

Fetch Fields(#SURNAME #GIVENAME #SALARY) From\_File(PSLMST) W Signal Event(uEmployeeFound) Uresult(#io\$sts)

Endroutine

Mthroutine Name(uClose) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(4) Name('uClose') Invoke Method(#COM\_OWNER.CloseForm) Endroutine

End\_Com

↑ Case 1 - Show a Form

#### Case 1 - Visual Basic - Object

```
VERSION 5.00
Begin VB.Form Form1
            = "Case 1 - VB"
 Caption
 ClientHeight = 4425
 ClientLeft
             = 60
 ClientTop
             = 345
 ClientWidth = 5835
 LinkTopic
             = "Form1"
              = 4425
 ScaleHeight
 ScaleWidth
              = 5835
 StartUpPosition = 3 'Windows Default
 Begin VB.CommandButton cmdClose
   Caption
              = "Close Visual LANSA"
              = 0 'False
   Enabled
   Height
              = 375
   Left
            = 4320
             = 10
   TabIndex
   Top
             = 3480
   Width
              = 1095
 End
 Begin VB.CommandButton cmdUpdate
              = "Update"
   Caption
              = 0 'False
   Enabled
   Height
              = 375
   Left
            = 4320
   TabIndex
               = 9
   Тор
             = 2760
   Width
             = 1095
 End
 Begin VB.Frame frmEmployeeDetails
              = "Employee Details"
   Caption
   Height
              = 2295
            = 120
   Left
   TabIndex
               = 3
             = 1680
   Top
```

Width = 3975 Begin VB.TextBox UemployeeSalary BeginProperty DataFormat Type = 1 = "0" Format HaveTrueFalseNull= 0 FirstDayOfWeek = 0FirstWeekOfYear = 0 LCID = 3081 SubFormatType = 1 EndProperty Height = 350 Left = 1680 TabIndex = 6 = "Salary" Text Тор = 1560 Width = 1215 End Begin VB.TextBox UemployeeGiveName Height = 350 Left = 1680TabIndex = 5 = "Give Name" Text = 360 Top Width = 1815 End Begin VB.TextBox UemployeeSurname = 345 Height Left = 1680 TabIndex = 4 Text = "Surname" Top = 960 Width = 1815 End Begin VB.Label lblSalary = "Salary:" Caption Height = 345 Left = 240 TabIndex = 8

= 1560 Top Width = 855 End Begin VB.Label LblName Caption = "Name:" Height = 345 Left = 240 TabIndex = 7 = 360 Top = 855 Width End End Begin VB.TextBox uEmployeeNumber = 350 Height = 1920 Left = 1 TabIndex = "A1012" Text = 1080Top Width = 735 End Begin VB.CommandButton cmdShowEmployee = "Show" Caption = -1 'True Default = 350 Height = 4320 Left = 0 TabIndex ToolTipText = "Invoke Visual LANSA method to retrieve employee de Top = 2040 Width = 1095 End Begin VB.Label lblexplanation = "Demonstrates the essentials of how to expose properties, Caption **BeginProperty Font** Name = "MS Sans Serif" Size = 9.75 Charset = 0 Weight = 400 Underline = 0 'False Italic = 0 'False

```
Strikethrough = 0 'False
   EndProperty
              = 495
   Height
   Left
            = 240
   TabIndex
             = 11
   Тор
             = 240
   Width
              = 5295
 End
 Begin VB.Label LblEmployee
              = "Employee Number:"
   Caption
   Height
              = 345
            = 360
   Left
   TabIndex
               = 2
             = 1080
   Тор
   Width
              = 1335
 End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
```

↑ Case 1 - Show a Form

#### Case 1 - Visual Basic - Code

Option Explicit Public Session As LANSA\_ACTIVEX\_LIB.Session

Private Sub AXFORMA\_uEmployeeFound(ByVal uResult As String) On Error GoTo ErrorHandler

If uResult = "OK" Then cmdUpdate.Enabled = True cmdClose.Enabled = True UemployeeGiveName = AXFORMA.UemployeeGiveName UemployeeSurname = AXFORMA.UemployeeSurname UemployeeSalary = AXFORMA.UemployeeSalary Call AXFORMA.ShowForm Else UemployeeGiveName = "Not Found" UemployeeSurname = "Not Found" UemployeeSalary = 0 End If

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description) End Sub

Private Sub cmdClose\_Click()

Call AXFORMA.uClose ' Unload AXFORMA

End Sub

Private Sub cmdShowEmployee\_Click() On Error GoTo ErrorHandler

' pass the employee number to the vl component method.

Call AXFORMA.uShowEmployee(uEmployeeNumber)

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description) End Sub

Private Sub cmdUpdate\_Click()

```
AXFORMA.UemployeeGiveName = UemployeeGiveName

' notice surname has been set as readonly in the Visual LANSA component

' AXFORMA.UemployeeSurname = UemployeeSurname

If UemployeeSalary = "" Then UemployeeSalary = 0

AXFORMA.UemployeeSalary = UemployeeSalary
```

End Sub

Private Sub Form\_Load() On Error GoTo ErrorHandler

' login to LANSA using default user, password and session location
If Session Is Nothing Then
Call ConnectToLansa("<user name>", "<password>", "
<session.cfg path>")
End If
' now add your component to the current session
Call Session.AddComponent(AXFORMA.object)

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description) End Sub

Private Sub ConnectToLansa(ByVal username As String, ByVal password As 5 On Error GoTo ErrorHandler

Set Session = New LANSA\_ACTIVEX\_LIB.Session

' Set the session configuration file Session.ConfigFile = txtlocation

Call Session.SetConnectParam("USER", username) Call Session.SetConnectParam("PSPW", password)

Call Session.Connect

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description) End Sub

Private Sub Form\_Unload(Cancel As Integer)

Set Session = Nothing

End Sub

↑ Case 1 - Show a Form

# **Case 2 - Embed a Reusable Part**

ployee Numb	er: 🛛	1012 Show Skills
Skill	Grade	Skill Full Description
ADMIN1	D	Administratn Part 1
ADMIN2	Р	Administratn Part 2
CS	Р	Computer Science Deg
HIS	Р	HISTORY DEG
INTRO	P	COMPANY INTRODUCTION
KEY	Р	KEYBOARD SKILLS
MANAGE1	D	Management Course 1
MANAGE3	F	Management Course 3
MARKET1	D	Marketing Course 1
OTH	D	Other Degree Course
oloyee skills si	alactad	

For code samples, go to:

Case 2 - Visual LANSA Source

Case 2 - Visual Basic - Object

Case 2 - Visual Basic - Code

↑ 7.3.6 Source Code

### Case 2 - Visual LANSA Source

```
Function Options(*DIRECT)
BEGIN_COM ROLE(*EXTENDS #PRIM_PANL) DISPLAYPOSITION(1) H
Attribute Class(#PRIM ATTR.AX TYPELIB) Guid('{0D4F2AF5-A003-
464A-A10E-C60842AB9A4E}') TypeLibName('LANSA AXPANELA LIB')
Attribute Class(#PRIM ATTR.AX CLASS) Guid('{643DEA7B-A45C-
498D-9983-F5044CE8281E}') ProgId('LANSA.AXPANELA')
Attribute Class(#PRIM ATTR.AX IN INTERFACE) Guid('{AD3DAC73-
0CE5-44BE-8694-740FFF1E78B}') BaseDispId(0)
Attribute Class(#PRIM ATTR.AX EVT INTERFACE) Guid('{7BDC12E6-
3D78-4247-A89F-FF763D0DB017}') BaseDispId(0)
DEFINE COM CLASS(#PRIM GRID) NAME(#GRID 1) CAPTIONNOBL
DEFINE COM CLASS(#PRIM ATLM) NAME(#ATLM 1)
DEFINE COM CLASS(#PRIM ATLI) NAME(#ATLI 1) ATTACHMENT(C
DEFINE COM CLASS(#PRIM GDCL) NAME(#GDCL 1) CAPTIONALIG
DEFINE COM CLASS(#PRIM GDCL) NAME(#GDCL 2) CAPTIONTYPE
DEFINE COM CLASS(#PRIM GDCL) NAME(#GDCL 3) CAPTIONALIG
DEFINE COM CLASS(#PRIM GDCL) NAME(#GDCL 4) PARENT(#GRII
```

Define\_Pty Name(uEmployeeGiveName) Get(\*auto #givename) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(0) Name('uEmploy Define\_Pty Name(uEmployeeSurname) Get(\*auto #surname) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(1) Name('uEmploy Define\_Pty Name(uEmployeeSalary) Get(\*auto #salary) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(2) Name('uEmploy \*

Mthroutine Name(uShowEmployeeSkills) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(3) Name('uShowEi Define\_Map For(\*input) Class(#empno) Name(#i\_empno) Define\_Map For(\*output) Class(#prim\_boln) Name(#uDirty)

Set Com(#uDirty) Value(#com\_false)

Clr\_List Named(#GRID\_1) Change Field(#EMPNO) To(#I\_EMPNO) Fetch Fields(#SURNAME #GIVENAME #SALARY) From\_File(PSLMST) W If\_Status Is(\*OKAY) Select Fields(#SKILCODE #GRADE) From\_File(PSLSKL) With\_Key(#EMP) Fetch Fields(#SKILDESC) From\_File(SKLTAB) With\_Key(#SKILCODE) Ke Add\_Entry To\_List(#GRID\_1) Endselect Endif \* set return code If Cond('#grid\_1.entries \*eq 0') Set Com(#uDirty) Value(#com\_true) Endif

Endroutine

End\_Com

1 Case 2 - Embed a Reusable Part

#### Case 2 - Visual Basic - Object

```
VERSION 5.00
Object = "{831FDD16-0C5C-11D2-A9FC-
0000F8754DA1}#2.0#0"; "MSCOMCTL.OCX"
Begin VB.Form Form1
 Caption
            = "Imbedded version of ActiveX Interaction 101"
 ClientHeight = 4545
 ClientLeft
            = 60
             = 345
 ClientTop
 ClientWidth = 5400
 LinkTopic = "Form1"
 ScaleHeight = 4545
 ScaleWidth
              = 5400
 StartUpPosition = 3 'Windows Default
 Begin MSComctlLib.StatusBar sbcase2
   Align
             = 2 'Align Bottom
             = 375
   Height
            = 0
   Left
   TabIndex = 3
   Тор
            = 4170
   Width
            = 5400
   _ExtentX = 9525
_ExtentY = 661
             = 1
   Style
              = 393216
   Version
   BeginProperty Panels {8E3867A5-8586-11D1-B16A-00C0F0283628}
    NumPanels
                  = 1
    BeginProperty Panel1 {8E3867AB-8586-11D1-B16A-00C0F0283628}
    EndProperty
   EndProperty
 End
 Begin VB.CommandButton cmdShowEmployee
              = "Show Skills"
   Caption
   Default
              = -1 'True
           = 350
   Height
   Left
            = 3000
```

```
TabIndex
             = 1
   ToolTipText = "Invoke Visual LANSA method to retrieve employee de
   Тор
             = 120
   Width
             = 1455
 End
 Begin VB.TextBox uEmployeeNumber
   Height
              = 350
   Left
            = 1680
   TabIndex
              = 0
            = "A1012"
   Text
   Тор
             = 120
   Width
             = 975
 End
 Begin VB.Label LblEmployee
              = "Employee Number:"
   Caption
   Height
              = 345
   Left
            = 120
   TabIndex
              = 2
             = 120
   Top
   Width
         = 1335
 End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
```

↑ Case 2 - Embed a Reusable Part

### Case 2 - Visual Basic - Code

Option Explicit Public Session As LANSA\_ACTIVEX\_LIB.Session Public gbDirty As Boolean

Private Sub cmdShowEmployee\_Click()

```
Call AXPANELA.uShowEmployeeSkills(uEmployeeNumber, gbDirty)
If gbDirty = True Then
sbcase2.SimpleText = "No skills found for employee"
Else:
sbcase2.SimpleText = "Employee skills selected"
End If
```

End Sub

```
Private Sub Form_Load()
```

```
'login to LANSA using default user, password and session location
If Session Is Nothing Then
Call ConnectToLansa("<user name>", "<password>", "
<session.cfg path>")
End If
```

```
Call Session.AddComponent(AXPANELA.object)
```

End Sub

Private Sub ConnectToLansa(ByVal username As String, ByVal password As On Error GoTo ErrorHandler

```
Set Session = New LANSA_ACTIVEX_LIB.Session
```

' Set the session configuration file Session.ConfigFile = txtlocation Call Session.SetConnectParam("USER", username) Call Session.SetConnectParam("PSPW", password)

Call Session.Connect

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description) End Sub

↑ Case 2 - Embed a Reusable Part

## **Case 3 - Using Components as Business Objects**

Department:	ADM	Search
Surname	Department	Given N.
BLAKE BLOGGS BROWN BROWN	MIS FLT ADM INF	JOHN FRED JO FREDD'- VERONI
Dickens FELL	LEG ADM	John ADAM P
Haliwell JACKSON Jones LEWIS LINCOLN	ADM SD ADM GAC INF	PATRIC STEPHE Benny TONY PAUL
▲ (	CAC	
В	lack F	orward
	ails	
Employee Deta		
Number:	A1012	_
	A1012 PATRICK	
Number:	J	
Number:	PATRICK	

For code samples, go to:

Case 3 - Visual LANSA Source Reusable Part AXOBJECTA Reusable Part AXOBJECTB Case 3 - Visual Basic - Object

Case 3 - Visual Basic - Code

↑ 7.3.6 Source Code

### **Case 3 - Visual LANSA Source**

Reusable Part AXOBJECTA

Reusable Part AXOBJECTB

↑ Case 3 - Using Components as Business Objects

#### Case 3 - Visual Basic - Object

```
VERSION 5.00
Begin VB.Form Form1
            = "Case 3 - VB"
 Caption
 ClientHeight = 8325
 ClientLeft
             = 60
 ClientTop
             = 345
 ClientWidth = 4650
             = "Form1"
 LinkTopic
              = 8325
 ScaleHeight
 ScaleWidth
              = 4650
 StartUpPosition = 3 'Windows Default
 Begin MSComctlLib.ListView lvwemployees
   Height
             = 3015
   Left
            = 360
   TabIndex
              = 15
   Тор
            = 840
   Width
             = 3975
   ExtentX
              = 7011
               = 5318
   ExtentY
   View
             = 3
               = -1 'True
   LabelWrap
   HideSelection = -1 'True
   FullRowSelect = -1 'True
   Version = 393217
   ForeColor
               = -2147483640
   BackColor
               = -2147483643
  BorderStyle
               = 1
   Appearance
               = 1
   NumItems
               = 0
 End
 Begin VB.ComboBox StatusMsgBox
   BackColor
               = &H8000013&
  Height
             = 315
            = 120
   Left
            = 2 'Dropdown List
   Style
```

TabIndex = 14 = 7920 Тор Width = 4455 End Begin VB.CommandButton cmdback = "Back" Caption = 0 'False Enabled = 375 Height Left = 1080 = 1 'Graphical Style = 13 TabIndex = 4080 Top Width = 1095End Begin VB.CommandButton cmdforward = "Forward" Caption = 0 'False Enabled = 375 Height Left = 2520 = 1 'Graphical Style TabIndex = 12 Top = 4080 Width = 1095 End Begin VB.Frame frmEmployeeDetails = "Employee Details" Caption Height = 3015 Left = 360 = 3 TabIndex Тор = 4680 Width = 3975 Begin VB.CommandButton cmdUpdate = "Update" Caption Enabled = 0 'False Height = 375 Left = 1560 TabIndex = 11 Тор = 2400 Width = 1095

End Begin VB.TextBox uSalary **BeginProperty DataFormat** Type = 1 = "0.00" Format HaveTrueFalseNull= 0 FirstDayOfWeek = 0FirstWeekOfYear = 0= 3081 LCID SubFormatType = 1 EndProperty Height = 350 Left = 1560 MaxLength = 11 TabIndex = 7 = "Salary" Text Тор = 1800 Width = 1215 End Begin VB.TextBox uGiveName = 350 Height Left = 1560 TabIndex = 6 = "Give Name" Text Top = 840 Width = 1815 End Begin VB.TextBox uSurname Height = 345 Left = 1560 TabIndex = 5 Text = "Surname" = 1320 Top Width = 1815 End Begin VB.TextBox uNumber BackColor = &H80000013& Enabled = 0 'False Height = 350

= 1560 Left TabIndex = 4 TabStop = 0 'False = 360 Top Width = 1215 End Begin VB.Label lblSalary = "Salary:" Caption Height = 345 Left = 240 TabIndex = 10 Top = 1800 Width = 855End Begin VB.Label LblName Caption = "Name:" Height = 225 = 240 Left TabIndex = 9 Тор = 840 Width = 855 End Begin VB.Label lblnumber Caption = "Number:" Height = 375 Left = 240 TabIndex = 8 = 360 Тор Width = 735 End End Begin VB.CommandButton cmdSearch = "Search" Caption Default = -1 'True Height = 375 Left = 2880 TabIndex = 2 Тор = 240 Width = 1215

End Begin VB.TextBox uDepartment Height = 350 Left = 1800 TabIndex = 1 = "ADM" Text = 240 Тор Width = 495 End Begin VB.Label lblDepartment Caption = "Department:" Height = 350 Left = 480 TabIndex = 0 Тор = 240 Width = 1215 End End Attribute VB\_Name = "Form1" Attribute VB\_GlobalNameSpace = False Attribute VB\_Creatable = False Attribute VB\_PredeclaredId = True Attribute VB\_Exposed = False

↑ Case 3 - Using Components as Business Objects

#### Case 3 - Visual Basic - Code

Option Explicit Public Session As LANSA\_ACTIVEX\_LIB.Session Dim axobjecta As LANSA\_AXOBJECTA\_LIB.axobjecta Dim axobjectb As LANSA\_AXOBJECTB\_LIB.axobjectb

Dim gbDirty As Boolean Dim gbDepartment As String \* 3 Dim lvwindex As Long

```
Private Sub Form_Load()
```

' login to LANSA using default user, password and session location If Session Is Nothing Then

Call ConnectToLansa("<user name>", "<password>", " <session.cfg path>") End If

```
Set axobjecta = Session.CreateComponent("AXOBJECTA")
Set axobjectb = Session.CreateComponent("AXOBJECTB")
```

' dynamically create columns for list view MakeColumns

```
GetEmployees ("")
```

End Sub

Private Sub GetEmployees(ByVal gbDepartment)

StatusMsgBox.Clear StatusMsgBox.AddItem ("Building Employee List") ' build collection of all employees for initial display If gbDepartment = "" Then ' build collection of all employees Call axobjectb.uGetAllEmployees(gbDirty) Else

Call axobjectb.uGetDepartEmployees(gbDepartment, gbDirty) End If

StatusMsgBox.Clear If UCase(gbDirty) = True Then ' if returned in error StatusMsgBox.AddItem ("Error when building list") Else: NewAddListItems ' add items to list StatusMsgBox.AddItem ("Employees selected")

End If

End Sub

Private Sub cmdSearch\_Click()

cmdback.Enabled = False cmdforward.Enabled = False

StatusMsgBox.Clear

StatusMsgBox.AddItem ("Building Employees for Department")
' build collection of all employees for initial display
Call axobjectb.uGetDepartEmployees(uDepartment, gbDirty)
If gbDirty = True Then
StatusMsgBox.AddItem ("Department not found")
' clear list view
 lvwemployees.ListItems.Clear
Else:
 ' add items to list
 NewAddListItems ' add items to list
StatusMsgBox.AddItem (Str\$(axobjectb.uEmployees.ItemCount) + " Emplo
End If

End Sub

Private Sub NewAddListItems() Dim employeeli As ListItem Dim employeesi As ListSubItem Dim employeeobject As LANSA\_AXOBJECTA\_LIB.axobjecta

On Error GoTo ErrorHandler

' clear list view lvwemployees.ListItems.Clear

gbDirty = axobjectb.uGetNextEmployee

```
Do While gbDirty = False
lvwindex = lvwindex + 1
Set employeeobject = axobjectb.uCurrentEmployee
Set employeeli = lvwemployees.ListItems.Add(, _
employeeobject.uNumber)
Call employeeli.ListSubItems.Add(, "surname", employeeobject.uSurname)
Call employeeli.ListSubItems.Add(, "department", employeeobject.uDepartu
Call employeeli.ListSubItems.Add(, "givenname", employeeobject.uGiveNa
Call employeeli.ListSubItems.Add(, "salary", employeeobject.uSalary)
gbDirty = axobjectb.uGetNextEmployee
Loop
```

```
Exit Sub 'Exit to avoid handler.
ErrorHandler: 'Error-handling routine.
MsgBox ("Error :" + Err.Description)
End Sub
```

Private Sub lvwemployees\_ItemClick(ByVal Item As ListItem) On Error GoTo ErrorHandler Dim employeeli As ListItem

cmdback.Enabled = True cmdforward.Enabled = True cmdUpdate.Enabled = True

```
Set employeeli = lvwemployees.SelectedItem
lvwindex = Item.Index
Call GetEmployee(lvwindex)
```

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description)

End Sub

Private Sub GetEmployee(lvwindex) Dim employeeobject As LANSA\_AXOBJECTA\_LIB.axobjecta

On Error GoTo ErrorHandler

Call axobjectb.uSetCurrentEmployee(lvwindex) Set employeeobject = axobjectb.uCurrentEmployee

uNumber = employeeobject.uNumber uSurname = employeeobject.uSurname uGiveName = employeeobject.uGiveName uSalary = employeeobject.uSalary

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description)

End Sub

```
Private Sub cmdback_Click()
```

If lvwindex > 1 Then lvwindex = lvwindex - 1 GetEmployee (lvwindex) End If

End Sub

Private Sub cmdforward\_Click()

```
If lvwindex < lvwemployees.ListItems.Count Then
lvwindex = lvwindex + 1
End If
```

GetEmployee (lvwindex)

End Sub

Private Sub cmdUpdate\_Click() Dim Status As String Dim employeeobject As LANSA\_AXOBJECTA\_LIB.axobjecta

On Error GoTo ErrorHandler

```
If uSalary = "" Then uSalary = 0
  Set employeeobject = axobjectb.uCurrentEmployee
  With employeeobject
    .uGiveName = uGiveName
    .uSurname = uSurname
    .uSalary = uSalary
  End With
  Status = employeeobject.uSaveEmployee
  ' Clear the Messages from the Message Combo
  StatusMsgBox.Clear
  If (Status = "OK") Then
    MsgBox ("Successfully updated " + employeeobject.uNumber)
  Else
    Call ShowMessages(Status, employeeobject)
    MsgBox ("Error Updating " + employeeobject.uNumber + " : " + Status)
  End If
Exit Sub
           'Exit to avoid handler.
```

ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description) End Sub

Private Sub ShowMessages(ByRef Status As String, ByRef employeeobject As Dim msgField As Object On Error GoTo ErrorHandler

For Each msgField In employeeobject.uMessages StatusMsgBox.AddItem (msgField.Value()) Next

```
' Select the first entry in the combo box
  If (StatusMsgBox.ListCount = 0) Then
    StatusMsgBox.AddItem ("Status = "' + Status + "'")
  End If
  StatusMsgBox.ListIndex = 0
Exit Sub
           ' Exit to avoid handler.
ErrorHandler: 'Error-handling routine.
  MsgBox ("Error :" + Err.Description)
End Sub
Private Sub MakeColumns()
 ' Clear the ColumnHeaders collection.
 lvwemployees.ColumnHeaders.Clear
 ' Add four ColumnHeaders.
 lvwemployees.ColumnHeaders.Add,, "", 0
 lvwemployees.ColumnHeaders.Add,, "Surname", 1500
 lvwemployees.ColumnHeaders.Add , , "Department"
 lvwemployees.ColumnHeaders.Add , , "Given Name"
 lvwemployees.ColumnHeaders.Add,, "Salary"
```

lvwemployees.View = lvwReport

End Sub

Private Sub ConnectToLansa(ByVal username As String, ByVal password As S On Error GoTo ErrorHandler

Set Session = New LANSA\_ACTIVEX\_LIB.Session

' Set the session configuration file Session.ConfigFile = txtlocation

Call Session.SetConnectParam("USER", username) Call Session.SetConnectParam("PSPW", password)

Call Session.Connect

Exit Sub 'Exit to avoid handler. ErrorHandler: 'Error-handling routine. MsgBox ("Error :" + Err.Description) End Sub

↑ Case 3 - Using Components as Business Objects

# 8. .NET Components

.NET Components are third-party reusable components which have been created using the Microsoft .NET Framework. .NET Components may be components or controls that you have created yourself, purchased from a third-party, or standard controls in the Microsoft .NET Framework.

Examples of .NET Framework components are calendars, graphing, spell checking, advanced grids and browsers.

LANSA provides support for the use of .NET Framework components and controls in Visual LANSA projects.

# 8.1 Before You Decide to Use a .NET Component

When you decide to use a .NET Framework component within your LANSA application you need to know that:

- You are introducing a dependency into your application over which the LANSA product has no control. Such a dependency may not be obvious or may not appear until sometime in the future.
- LANSA support is not obliged to help you resolve any problem with .NET Framework components. They may or may not have information available to assist you.
- Any problem you report that involves a .NET Framework component must be reproducible on your development Visual LANSA system. Problems related to .NET Framework components deployed to non-development systems should not be reported.

## 8.2 Using .NET Components

To use a .NET Framework component, first ensure that the computer has the Microsoft .NET Framework turned on. Versions of Microsoft Windows since Windows Vista/Windows 2008 come with the .NET Framework included but not all versions of the framework are turned on by default. See Microsoft Windows documentation for further help on turning on the .NET Framework.

The next step is to create a .NET Component in Visual LANSA. Once created, the .NET Component can be used the same way as any other Visual LANSA component. An instance of the component can be created and its properties, methods and events can be used. Some components are very simple and little programming is required to use them, others provide complex functionality and may require much more coding than using native Visual LANSA controls.

### 8.2.1 Obtaining .NET Framework Components

.NET Framework components can be purchased separately from a variety of vendors. Many of them can be obtained from Web sites and you can usually download them for evaluation.

Alternatively, you may choose one of the standard components in the .NET Framework. For example, the WebBrowser control. You may also develop your own .NET component. This will require a tool such as Microsoft's Visual Studio or similar.

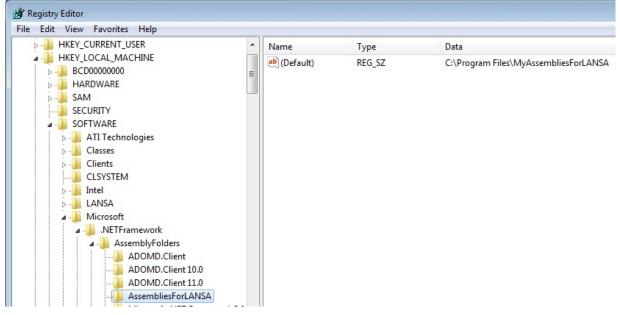
## **8.2.2 Enrolling a .NET Framework Component in the Repository**

To use a .NET Framework component, the .NET assembly which contains the component must be enrolled in the Repository as a .NET Component. The .NET assembly is normally a DLL file. For Visual LANSA to locate the .NET assembly you may specify the fully qualified file name or you may register the assembly location in the standard Microsoft location.

To register the assembly location, you must modify the Windows registry. Follow these steps:

- 1. Open the Registry Editor. Run regedit.exe from the Windows Start menu.
- On a 32-bit system navigate to HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\.NETFramework\Assen On a 64-bit system navigate to HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETF1
- 3. Create a new key. The name you assign this key will be used in the .NET Component.
- 4. Change the value of the (Default) value. The value should be the fully qualified path of the directory where your assembly is located. This path may contain multiple assemblies.

Care should always be taken when modifying the Windows Registry.



To create a .NET Component in the Visual LANSA Repository, create a new component of type .NET Component. The new .NET Component will contain

an empty Componentassembly property.

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_OBJT) Componentassembly(") End\_Com

Select the *Details* in the LANSA Editor.

**Method 1:** Type the fully qualified name into the Componentassembly property.

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT) Componentassembly('c:\Program

**Method 2:** Type the Assembly name and the registry key name which contains the path to the assembly. The two values should be comma separated.

Begin\_Com Role(\*EXTENDS #PRIM\_OBJT) Componentassembly('MyAssemb

Alternatively, use the prompter (---) in the Details view. The prompter will list all assemblies that are found in the locations pointed to by all keys under the AssemblyFolders registry key described in step 2.

Select the relevant assembly and the prompter will set the

Componentassembly property appropriately. To specify the assembly using a fully qualified name, click the *Browse for Assembly File* button. This will display the *Open* dialog. Navigate to the directory which contains the assembly. Select the assembly, click *Open* and the prompter will set the

Componentassembly property appropriately.

## 8.2.3 Using a .NET Component

The .NET Component can be used the same way as any other Visual LANSA component.

## Define an instance of a component.

Use the DEFINE\_COM command. If the class does not have a constructor or has a constructor which takes no parameters then you may define the instance as either static or dynamic. Otherwise you must define the instance as dynamic.

Define\_Com Class(#DOTNET1.TestClasses.MyClass) Name(#MyClass) Referer

If the class is a control, be sure to parent the control to the form so that it is visible on the form. You may then resize and move the control in Design view.

Define\_Com Class(#DOTNET1.DotNetControls.WebBrowserX) Name(#Browse

## Instantiating a dynamic instance

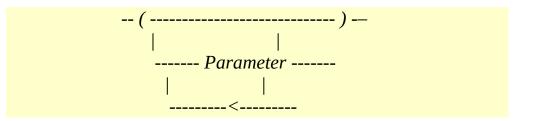
If the component instance has been defined as dynamic, the instance will need to be **instantiated** with either the \*New or SET\_REF commands. Either command allows you to specify a constructor.

## How to define classes with static methods

To use static methods in a class you must define a component instance. The instance may be defined as static or dynamic. Once defined, static methods from the class may be called at any time. If a dynamic component instance has been defined, static methods may be called without instantiating the instance. This allows the use of static classes and also allows the instantiation of classes with private or protected constructors (assuming the class has static methods to instantiate itself).

## \*NEW

If the class does not have a constructor or has a constructor that takes no parameters, then you may specify no constructor or a constructor with no parameters. Otherwise the parameters should be passed similar to a method call.



## SET\_REF

For information, refer to the SET\_REF in the *Technical Reference Guide*.

#### Examples

#### C#

public ConstructorTests() {...}
public ConstructorTests( int aNumber ) {...}
public ConstructorTests( int aNumber, ref DateTime aDateTime, String aString

#### RDML

Define\_Com Class(#DOTNET1.TestClasses.ConstructorTests) Name(#LCom0

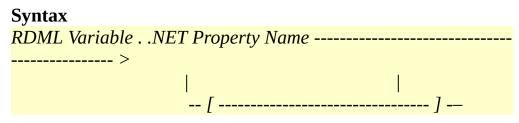
Define Field(#LNum01) Type(\*INT) Length(4) Define Field(#LStr01) Type(\*CHAR) Length(50) Define Field(#LDTime01) Type(\*DATETIME) Length(26)

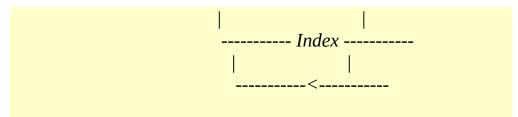
#LCom01 <= \*New #DOTNET1.TestClasses.ConstructorTests Set\_Ref Com(#LCom01) To(\*CREATE\_AS #DOTNET1.TestClasses.Construc

#LCom01 <= \*New #DOTNET1.TestClasses.ConstructorTests.ConstructorTest Set\_Ref Com(#LCom01) To(\*CREATE\_AS #DOTNET1.TestClasses.Construc

#LCom01 <= \*New #DOTNET1.TestClasses.ConstructorTests.ConstructorTest Set\_Ref Com(#LCom01) To(\*CREATE\_AS #DOTNET1. TestClasses.Construct

## Using properties and variables





RDML Variable . .NET Variable Name

#### Examples

```
C#

public int Counter

{

get { return mCount; }

set { mCount = value; }

}
```

#### RDML

Define\_Com Class(#DOTNET1.TestClasses.MyClass) Name(#NumericClass)

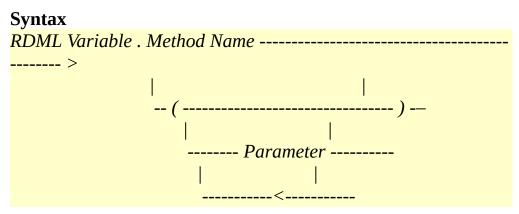
```
#NumericClass.Counter := 120
```

#STD\_INT := #NumericClass.Counter

Begin\_Loop Using(#STD\_INT) To(#NumericClass.Counter) Step(2)

... End\_Loop

## **Calling Methods**



Example

```
C#

public void

IncrementDateByDays(

ref DateTime dateTime,

int dayAdjustment )

{

dateTime = dateTime.AddDays( dayAdjustment );

}
```

#### RDML

```
Define_Com Class(#DOTNET1.TestClasses.MyClass) Name(#LClass01)
Define Field(#LDat01) Type(*DATETIME)
Define Field(#LDat02) Type(*DATETIME)
```

```
#LDat01 := '1954-04-25 10:04:00.000000'
#LDat02 := '1954-05-05 10:04:00.000000'
```

```
#LClass01.IncrementDateByDays( #LDat01 10 )
```

```
If ((#LDat01 = #LDat02))
```

.. Endif

```
C#

public String

DoubleIntegerReturnString(

int number,

out int result )

{

result = number * 2;

return String.Format( "Double {0} is {1}", number, result );

}

RDML

Define_Com Class(#DOTNET1.TestClasses.MyClass) Name(#LClass01)

Define_Distribution (#DMT) Less (#DOTNET1.TestClasses.MyClass) Name(#LClass01)
```

```
Define Field(#LNum01) Type(*INT) Length(4)
Define Field(#LNum02) Type(*INT) Length(4)
Define Field(#LStr01) Type(*CHAR) Length(80)
```

```
#LNum01 := 411
#LStr01 := #LClass4101.DoubleIntegerReturnString( #LNum01 #LNum02 )
If (#LNum02 = (#LNum01 * 2))
...
Endif
```

## **Event Handling**

Event handling functions are defined using the EVTROUTINE command. See the EVTROUTINE documentation in the *Technical Reference Guide* for further details about this command. The basic syntax of the command is as follows:

```
EVTROUTINE --- HANDLING ( Variable Name . Event Name )--
------>
```

#### Example

```
C#

public class EventArgs1

{

private String mName;

public EventArgs1() {

public String Name

{

get { return mName; }

set { mName = value; }

}

Public class EventTests

{

public event MyEventDelegate MyEvent;

}
```

```
public delegate void MyEventDelegate( String str, EventArgs1 eventArgs );
```

#### RDML

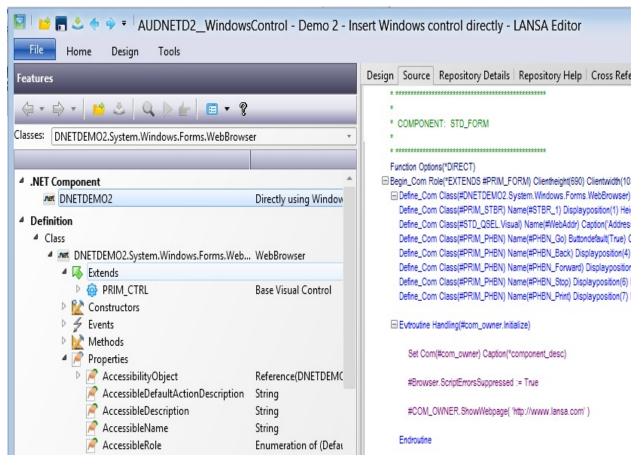
Define\_Com Class(#DOTNET1.TestClasses.EventTests) Name(#EventTest)

Evtroutine Handling(#EventTest.MyEvent) Str(#s1) EventArgs(#c1) #STD\_TEXT := #s1 #STD\_NAME := #c1.Name Endroutine

}

## **.NET Component Features**

As with native Visual LANSA controls, you can view a description of a .NET Component's classes, constructors, properties, methods and events by selecting it and selecting the Features view (F2).



The help provided for .NET Components through the Features help is brief. Help is only available if an XML file was generated with appropriate comments when the .NET assembly was created. If an XML file does exist it should be copied to the same location as the assembly. Double click on a constructor, property, method or event to see its help text (if help exists for the selected feature).

The 'Classes' drop down in the Feature Help lists all available classes in your .NET component. Select a class to view all its constructors, properties (both Visual LANSA and built-in), events and methods. The Visual LANSA properties are used to set attributes which are common to all controls such as their height and the width.

## **.NET Collections**

.NET Collections can accessed using the following methods:

## Visual LANSA For statement

Define Field(#LNum6501) Type(\*INT) Define Field(#LStr6501) Type(\*STRING) Length(20) Define Field(#LBol6501) Type(\*BOOLEAN)

Define\_Com Class(#DNE003\_SystemDll.System.Collections.Specialized.StringCollection) Name(#LCls6501\_Coll) Reference(\*DYNAMIC)

#LCls6501\_Coll <= \*New #DNE003\_SystemDll.System.Collections.Specialized.StringCollection

#LStr6501 := 'Test' #LCls6501\_Coll.Add( 'Test1' ) #LCls6501\_Coll.Add( 'Test2' ) #LCls6501\_Coll.Add( 'Test3' )

```
#LNum6501 := 0
#LBol6501 := true
For Each(#Current) In(#LCls6501_Coll)
#LNum6501 += 1
If (#Current <> (#LStr6501 + #LNum6501.AsString))
#LBol6501 := false
Endif
Endif
```

## .NET indexer (commonly represented by the Item property)

Define Field(#LNum6601) Type(\*INT) Define Field(#LStr6601) Type(\*STRING) Length(20) Define Field(#LBol6601) Type(\*BOOLEAN)

Define\_Com Class(#DNE003\_SystemDll.System.Collections.Specialized.StringCollection) Name(#LCls6601\_Coll) Reference(\*DYNAMIC)

```
#LCls6601_Coll <= *New
#DNE003_SystemDll.System.Collections.Specialized.StringCollection
```

#LStr6601 := 'Test'
#LCls6601\_Coll.Add( 'Test0' )
#LCls6601\_Coll.Add( 'Test1' )
#LCls6601\_Coll.Add( 'Test2' )

```
* A string collection - GET
#LBol6601 := False
If (#LCls6601_Coll.Item[2] = "Test2")
#LBol6601 := True
Else
#LBol6601 := False
Endif
```

```
* A string collection - SET
```

```
#LCls6601_Coll.Item[0] := 'NewValue0'
#LCls6601_Coll.Item[1] := 'NewValue1'
#LCls6601_Coll.Item[2] := 'NewValue2'
If (#LCls6601_Coll.Item[1] = 'NewValue1')
#LBol6601 := True
Else
#LBol6601 := False
Endif
```

#### .NET enumerator

Define Field(#LNum6401) Type(\*INT) Define Field(#LNum6402) Type(\*INT) Define Field(#LStr6401) Type(\*STRING) Length(20) Define Field(#LBol6401) Type(\*BOOLEAN)

Define\_Com Class(#DNE003\_SystemDll.System.Collections.Specialized.StringCollection) Name(#LClass6401\_Coll) Reference(\*DYNAMIC) Define\_Com Class(#DNE003\_SystemDll.System.Collections.Specialized.StringEnumerator) Name(#LClass6402\_Enum) Reference(\*DYNAMIC)

```
#LClass6401_Coll <= *New
#DNE003_SystemDll.System.Collections.Specialized.StringCollection
```

#LStr6401 := 'Test'
#LClass6401\_Coll.Add( 'Test1' )
#LClass6401\_Coll.Add( 'Test2' )
#LClass6401\_Coll.Add( 'Test3' )

```
* Use Enumerator
```

```
#LClass6402_Enum <= #LClass6401_Coll.GetEnumerator()
#LNum6401 := 0
#LBol6401 := true
#LClass6402_Enum.Reset()
Dowhile Cond(#LClass6402_Enum.MoveNext())
#LNum6401 += 1
If (#LClass6402_Enum.Current <> (#LStr6401 +
#LNum6401.AsString))
#LBol6401 := false
Endif
Endwhile
```

If (#LBol6401) #STD\_TEXT := 'Success' Else #STD\_TEXT := 'Failed' Endif

## .NET Component Sample Code

This sample uses the standard .NET assembly System.Windows.Forms.dll. This assembly is normally located in the following locations:

32-bit system: C:\Program Files\Reference

Assemblies\Microsoft\Framework\.NETFramework\v4.0

64-bit system: C:\Program Files (x86)\Reference

Assemblies\Microsoft\Framework\.NETFramework\v4.0

#### Create a .Net Component named #DNETDEMO

The resulting .Net Component source should look as follows (on 32-bit machines):

Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_OBJT) Componentassembly('C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.Windows.Foi End\_Com

#### Create a form with the following code

Function Options(\*DIRECT) Begin Com Role(\*EXTENDS #PRIM FORM) Clientheight(690) Clientwidth(1030) Height(728) Left(723) Top(181) Width(1046) Define\_Com Class(#DNETDEMO.System.Windows.Forms.WebBrowser) Name(#Browser) Displayposition(8) Height(617) Left(16) Parent(#COM\_OWNER) Tabposition(8) Top(40) Width(1001) Define\_Com Class(#PRIM\_STBR) Name(#STBR\_1) Displayposition(1) Height(24) Left(0) Messageposition(1) Parent(#COM\_OWNER) Tabposition(1) Tabstop(False) Top(666) Width(1030) Define\_Com Class(#STD\_QSEL.Visual) Name(#WebAddr) Caption('Address') Displayposition(2) Height(19) Labeltype(Caption) Left(96) Marginleft(60) Parent(#COM OWNER) Tabposition(2) Top(16) Usepicklist(False) Width(801) Define Com Class(#PRIM PHBN) Name(#PHBN Go) Buttondefault(True) Caption('Go') Displayposition(3) Left(904) Parent(#COM OWNER) Tabposition(3) Top(13) Width(27)

Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_Back) Displayposition(4) Image(#VB\_ARROWL) Left(22) Parent(#COM\_OWNER) Tabposition(4) Top(13) Width(27)

Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_Forward) Displayposition(5) Image(#VB\_ARROWR) Left(56) Parent(#COM\_OWNER) Tabposition(5) Top(13) Width(27) Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_Stop) Displayposition(6) Image(#VB\_STOP) Left(936) Parent(#COM\_OWNER) Tabposition(6) Top(13) Width(27) Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_Print) Displayposition(7) Image(#VB\_PRINT) Left(984) Parent(#COM\_OWNER) Tabposition(7) Top(13) Width(27)

Evtroutine Handling(#com\_owner.Initialize) Set Com(#com\_owner) Caption(\*component\_desc) #Browser.ScriptErrorsSuppressed := True #COM\_OWNER.ShowWebpage( 'http://www.lansa.com' ) Endroutine

Evtroutine Handling(#Browser.Navigated) Sender(#Sender) E(#EventArgs) #WebAddr := #Browser.Url.ToString() Endroutine

Evtroutine Handling(#PHBN\_Go.Click) If (#WebAddr.Substring( 1, 7 ).LowerCase() <> 'http://') #WebAddr := 'http://' + #WebAddr Endif

#COM\_OWNER.ShowWebpage( #WebAddr ) Endroutine

Evtroutine Handling(#PHBN\_Back.Click) #Browser.GoBack Endroutine

Evtroutine Handling(#PHBN\_Forward.Click) #Browser.GoForward Endroutine Evtroutine Handling(#PHBN\_Stop.Click) #Browser.Stop Endroutine

Evtroutine Handling(#PHBN\_Print.Click) #Browser.ShowPrintDialog Endroutine

Mthroutine Name(ShowWebpage) Define\_Map For(\*INPUT) Class(#STD\_QSEL) Name(#WebPageAddress) Define\_Com Class(#DNETDEMO.System.Uri) Name(#Uri) Reference(\*DYNAMIC)

#Uri <= \*New #DNETDEMO.System.Uri.Uri( #WebPageAddress ) #Browser.Url <= #Uri Endroutine

End\_Com

## 8.2.4 Deploying a Component

The .NET Framework assembly should be deployed with your Visual LANSA application using the LANSA Application Deployment Tool. For assistance in creating the deployment package see the LANSA Application Deployment Tool help guide.

This topic will concentrate on where to install the assembly on the target machine. The following lists the search order that Visual LANSA uses to locate the assembly.

- 1. The specified location. Whether you have specified the fully qualified path or used the registry method this is the first place that is searched. If you have used the registry entry method then you will need to create the registry entry on the deployed system. If you have used the fully qualified path then the assembly will need to be installed in the exact location on the deployed system.
- 2. Partition Execute directory (<sys dir>\x\_<ppp>\execute), for example <LANSA Root Directory>\x\_win95\x\_lansa\x\_<ppp>\execute.
- 3. System Execute directory (<sys dir>\execute), for example <LANSA Root Directory>\x\_win95\x\_lansa\ execute.
- 4. System wide generic load. Searches the Global Assembly Cache and the location of the running executable. Care should be taken when relying on this method because only the assembly name is used when searching for the assembly. Version, culture and the public key token are not specified. This means that you may inadvertently load the incorrect version of the assembly if there are other assemblies in the Global Assembly Cache with the same name.

Your .NET Framework assembly should be deployed to one of these locations.

# 8.3 Visual LANSA Restrictions and Naming Conventions for .NET Component Classes

Due to differences in syntax between RDML and .NET languages (such as C#), certain characters need to be replaced, naming conventions changed and restrictions enforced.

## **Generic Classes**

Generic Class names will have the generic type appended to their class name surrounded by '^'. For example,

```
C#
namespace TestClasses
{
public class GenericTests<T,Y>
}
```

#### RDML

 $\#DOTNET1.TestClasses.GenericTests^{\Lambda}Y^{\wedge}Y^{\wedge}$ 

## **Generic Methods**

Generic Methods are not supported.

## Arrays

Dimensions in array class names are represented by the following syntax

```
Class name ---- # ---- Number of dimensions ------|
| | |
```

For example:

```
C#
int[,][]
MyClass[]
```

RDML #DOTNET1.System.Int32#2#1 #DOTNET1.TestClasses.MyClass#1

## **Overloaded Functions**

Unique names are generated for overloaded functions. The first function found by the assembly parser will use the actual function name. All subsequent overloaded functions will use the actual function name suffixed by "#x" where *x* is an incrementing number starting at 1. For example, if a .NET Framework class has three functions named Add, the function names in RDML will be Add, Add#1 and Add#2. Use the Features view in the Visual LANSA editor to view the function signatures.

## LANSA Internal Names

The following names are used by internal Visual LANSA classes. You should avoid using these names in your .NET Framework components as the internal LANSA Property/Method/Event will take precedence.

Name	Owner	Parent
ComponentTag	ComponentType	ComponentMembers
ComponentTypeName	ComponentClassName	ComponentPatternName
Тор	Hint	HintTitle
Left	Focus	Width
Handle	Height	Parent
Enabled	TabStop	Visible
CanFocus	HintShow	Modified
DragStyle	PopupMenu	ScreenTop
ScreenLeft	SizingRule	BusyUpdates
VisualStyle	TabPosition	DisplayPosition
HintShowOfParent	BusyUpdatesOfParent	VisualStyleOfParent
Cursor	Opacity	OriginType
OriginX	OriginY	ScaleX
ScaleY	SkewX	SkewY

#### Properties

#### **Events**

CreateInstance	DestroyInstance	Realize
SetFocus	Unrealize	UpdateDisplay
ShowMessages		

#### Methods

EndDrag	DragDrop	DragOver
StartDrag	Initialize	MouseEnter
MouseLeave		

## 9. External Resources

A LANSA application consists of many objects, Forms, Files, Fields and so on. These objects are managed in the LANSA Repository.

Often there are 'other objects' that are a part of your application, such as Javascript, HTML-pages and images for web-development, or even simple configuration files for desktop-applications.

External Resources have been developed to manage these files. They can be registered in the Repository, checked into an IBM i master and deployed using the LANSA Deployment Tool.

External Resources are LANSA objects just like any other, but with the exception that the LANSA Editor doesn't know how to edit them—as it has no idea of its contents—and LANSA programs don't know how to use them—the code will still be just referring to a file on disk. We do, however, want you to register them in the Repository. This has two advantages: as a back up and to make it easier to correctly deploy them with your application.

#### Also see

9.1 Register Multiple External Resources9.2 Edit External Resources

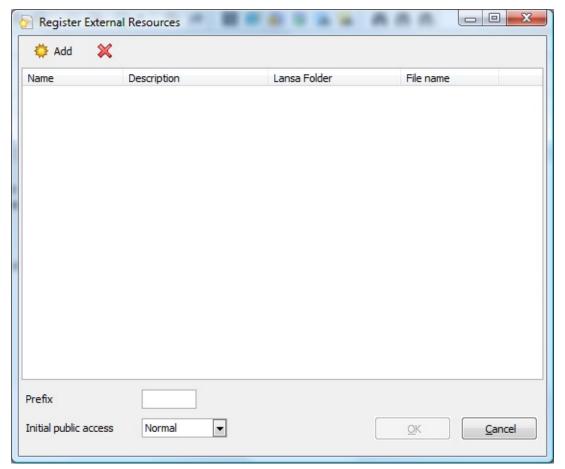
## 9.1 Register Multiple External Resources

There are two ways to register External Resources. Select the New option in the File menu, then External Resource and then you can either

• Choose External Resource and define it in the same way as you would define any other object

or

• Choose Bulk Loader option which will bring up the following dialog:



Use the Add button to select one or more files from your disk, or use Windows Explorer to drag and drop files into the Register External Resources dialog. Provide a prefix to name them all and press the *Create* button. Note that LANSA detects what folders the files are in.

## 9.2 Edit External Resources

The object in your repository is a LANSA object, therefore you must have the correct task and security settings to be able to edit it. Once it is open in the LANSA editor, you'll have an option to open the actual file in an external editor to edit its contents. When finished you must save the external resource in the LANSA editor to update its definition in the Repository.

Of course there is nothing to stop you from editing the file outside of LANSA. Beware though, that the file content that is Checked-in/Exported/Deployed is the one that is stored in the Repository, not the one on disk. So you will need to open and save the External Resource in the LANSA editor to deploy its latest version.

The *External Resource State* column in the Repository browser indicates the current state of the External Resource on disk versus the one in the Repository.

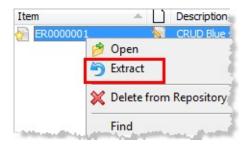
Item	A .	Description	Modified	Build Status	Details	Task
FR0000001	1	CRUD Blue	18/11/20		crud_blu	
		External Resour	co State			

The icons in this column have the following meaning:

*Blank* The local file no longer exists.

- The local file has been edited and is newer than the version in the Repository.
- The Repository version has changed and is newer than the local file.
- The local file and the Repository version are identical.

For instance, you will have an older version of the file than the one in the Repository if another developer has updated it and you check out the latest version. Check out will not replace your local files, neither will import or propagation. This is a manual process and you can update your local version using the *Extract* option on the context menu by right clicking on the object in the Repository list.



## ↑ 9. External Resources

## **10. Application Templates**

If you are a developer building applications with LANSA Application Templates, review the following:

**10.1 Using Application Templates** 

If you are a developer modifying or creating your own LANSA Application Templates, review the following:

**10.2 Creating Application Templates** 

10.3 Application Template Program Examples

#### Also See

Creating Applications using Functions

## **10.1 Using Application Templates**

10.1.1 What is a Template?

10.1.2 How are Templates Used?

10.1.3 LANSA Shipped Templates

10.1.4 Steps for Using Templates

10.1.5 Tips for Using Templates

10.1.6 Modifying Templates

10. Application Templates

## 10.1.1 What is a Template?

Application templates are one of the most powerful features within LANSA. A template can be thought of as an RDML code generator. It is an interpretive program written using its own language using 10.2.2 Template Commands. When executed, a template prompts the developer with a series of questions and then generates the appropriate RDML code based on the answers.

Templates are used with the LANSA Editor and can be used to add code to any LANSA object that stores RDML code. For example, templates can add code to functions, forms, reusable parts, or WAMs.

Templates are extremely important as they ensure consistency in function design, operation and structure. They are an efficient means of generating commercial grade applications, and can produce a complete Visual LANSA component, RDML function or just a block of RDML code.

LANSA also ships more than 40 ready-to-use templates to help you with application development. Specific online help is available for each shipped template. Because templates are themselves a "program", they can be created and changed by developers.

It is also very important to note that templates are defined at the system level of LANSA. A template is available to all partitions. If it is changed, all partitions will have the new changed template. Hence, it is always advisable to make a copy of a template and to never change the original template.

10.1 Using Application Templates

## **10.1.2 How are Templates Used?**

Templates can be used in two ways:

#### 1. To create a new function:

When a new function is being defined, there is an option to generate the function using an application template. Almost every new function should be created using a template to ensure that your function follows consistent coding standards.

#### 2. To add code to an existing LANSA object:

When modifying an existing LANSA object that contains RDML code, a template can be executed from within the editor by using the template icon on the toolbar.

Templates do not have to generate a complete program. In many cases, they are used as building blocks where you have common or repetitive code. For example, a building block template (BBPGM) can be used to produce a standard function header which can then be used with every new function. An IF/ENDIF template (BBIF) can be used to create the IF and ENDIF code in a function.

Where possible, it is recommended that you start with an application template or building block templates to create your functions.

Visual LANSA templates for component generation are prefixed with "VL\_". These templates are for component use only.

**Note:** Some specialized batch templates may be installed if you are using products such as Commerce Edition. These specialized templates are not typically for interactive execution in the LANSA editor. The template description identifies these specialized templates.

10.1 Using Application Templates

## **10.1.3 LANSA Shipped Templates**

LANSA ships a wide range of templates. Following is a partial list of the interactive templates to help you understand some ways that templates are used:

FRPOP01	Pop-up Prompter (Full File) Function
FRADD01	Flat Screen Data Entry Function
FRENQ01	Flat Screen Enquiry Function
FRMNT01	Flat Screen Maintenance Function
FRABMNT01	Action Bar Maintenance Function
FRPRT01	List Style Report Function
FRWRK01	Basic Work with Function
FRWEBADD01	Webevent Data Entry Function
BBIF	
BBIFELSE	RDML IF-ELSE-ENDIF statements
BBPGM	Online Program Layout (header)
BBPGMBCH	Batch Program Layout (Header)
BBFLDTRIG	Field Trigger Function
BBFILTRIG	File Trigger Function
BBCASE	Builds RDML CASE statement
BBPAGE	Page at a Time Browse List
VL_BASEBOB	Visual LANSA Business Object Browser
VL_BASEBOD	Visual LANSA Business Object Detailer
VL_BASEMNT	Visual LANSA Flat File Maintenance
VL_BASERTRE	Visual LANSA Simple Tree List
VL_BBMNUBR	Visual LANSA Menu Bar Building Block
and more	

It is strongly recommended that you try each of the interactive templates so that you are familiar with the building blocks that are available. (Specialized batch templates may exist for products such as LANSA's Commerce Edition. These templates are not for execution in the LANSA editor.)

Visual LANSA templates for component generation are prefixed with "VL\_".

10.1 Using Application Templates

## **10.1.4 Steps for Using Templates**

A template can be created for almost any code. Templates are generally created whenever you have a commonly used section of code or logic. Again, templates do not have to generate complete functions. They can create any standard block of code.

When using templates, you can follow five simple steps:

1. Review List of Templates

When creating a new function, you should first review the existing templates to see if you can use or modify the RDML they already provide.

2. Read Help Text

Each template comes with online help panels which explain what the template will produce and how you should answer the template questions.

3. Select Template

Once you have found a template which meets your needs, you can execute it when creating a function or when editing the LANSA object.

4. Execute Template

As you will notice in the help text documentation, templates support the use of the cancel key to move backwards through the interactive screens.

5. Compile & Test

Once the template has completed execution, the RDML commands will have been inserted into the LANSA object. This RDML can be modified or compiled as is (if a complete program was generated), and then tested.

It is strongly recommend that you review all the templates shipped with LANSA. You should copy and then modify the templates to meet your site standards.

10.1 Using Application Templates

## **10.1.5 Tips for Using Templates**

Following are some tips for using templates:

- When a question appears, read the question very carefully and completely. Review any additional prompting information very carefully. Be sure to follow instructions to the letter. If you have any questions use the HELP function key.
- Make extensive use of the HELP function key and take your time. When using a template for the first time, read all the HELP panels associated with the template, especially the examples.
- Not all templates use a "question and answer" session. If no question appears, and the "Process Definition Menu" is re-displayed, you can assume that the template has generated your RDML program without having to ask you any questions at all.
- The generalized nature of the template facility and its ability to be site definable, mean that absolutely precise validation of your answers is often not practicable.
- When lists of fields appear , there are generally two types of selection: ordered and non-ordered. Ordered selection will request a sequence number which actually determines the order of the fields in the function. Nonordered will just require a non-blank character to indicate that the field is selected in the list.
- Incorrect answers will cause no real problems. The RDML code created may not compile or may not execute correctly. You may need to delete the RDML and execute the template again or fix it by manually editing the RDML code.
- Most templates allow you to "back up" to a previously answered question and change your answer. To do this use the Previous/Cancel key. Do not use the cancel key if the prompt indicates that you should not.
- If you are not satisfied with the RDML generated, you may simply delete the generated code and try the template again.
- 10.1 Using Application Templates

## **10.1.6 Modifying Templates**

It is strongly recommended that you use LANSA templates. Consider the following:

- The template language is very easy to learn and use. It is a simple set of instructions based around a tag language.
- You can customize templates to meet your site standards. For example, you can decide if want your company name in the function header.
- You can create your own templates for almost any function. Any time you have repetitive code being used, it is a potential opportunity for a template.

## You can edit templates in Visual LANSA using the Template Maintenance option.

- If you are using a Slave Visual LANSA System, you can check-out templates for update and then check-in the changed template to LANSA for i.
- If you are using a Slave Visual LANSA System, you can use the Host Monitor Refresh to update the list of templates. The System Initialization option can also be used to retrieve the templates from the Master System.
- At the very minimum, you should customize and use the BBPGM template so that all templates have a consistent header and comment format.

#### Remember

- Templates are system wide. They are used in all partitions. Any changes will impact all partitions and all developers. When imported, they impact the whole system.
- Because templates are system wide, always make a copy and then modify the copy so that the original template is unchanged. This protects changes from being lost if someone re-imports a set of templates.

#### ↑ 10.1 Using Application Templates

## **10.2 Creating Application Templates**

The writing and maintenance of application templates should only be attempted by very experienced programmers who have extensive knowledge of both Application Template Commands and RDML Commands. The following topics provide an introduction to creating templates:

- 10.2.1 Introduction to Template Code
- 10.2.2 Template Commands
- 10.2.3 Template Special Variables

#### Also See

Application Templates in the LANSA Application Design Guide Template Commands and Variables in the Technical Reference Guide

10. Application Templates

## **10.2.1 Introduction to Template Code**

A template consists of three basic components:

- LANSA template commands
- LANSA RDML commands
- HELP text or panels.

An application template may be used either as a "skeleton" or "model" for the generation of a complete RDML program, or may be used to generate a commonly used section of RDML code or logic.

The application template commands control how, when, and what RDML commands are generated. They usually do this by asking the user a question, and by using the answer to generate different RDML commands.

The RDML commands produced are stored as part of the template. The template language controls which lines of code are written, and what values are used within the RDML commands.

The template HELP text is also stored as part of the template definition and is automatically displayed when the user presses the HELP key.

LANSA has a special 10.2.2 Template Commands language. All commands conform to the normal LANSA command conventions. For instance, positional or keyword parameter specification can be used, lists must be enclosed in brackets, etc. The template language is not compiled. It is an interpretive language.

All of LANSA's template commands begin with "@@". For example, "@@QUESTION" is an application template command which will ask the user a question, and receive and store the answer.

The RDML generated by a template is stored as part of the template. The "@@" template commands control the generation of the RDML code.

The application template commands and RDML code can also make use of 10.2.3 Template Special Variables. These variables will be substituted into the RDML code by LANSA before being added to the generated RDML program.

Reminder: A template definition exists across all partitions. It can be accessed and maintained from any partition, provided authority is granted. When deleted, it is deleted from all partitions.

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.

#### Also See

10.3 Application Template Program Examples

↑ 10.2 Creating Application Templates

## **10.2.2 Template Commands**

The LANSA template language is a simple "tag" language that relies on branch type logic. It is not a sophisticated programming language. All commands conform to the normal LANSA command conventions. For instance, positional or keyword parameter specification can be used, lists must be enclosed in brackets, etc. The template language is not compiled. It is an interpretive language.

**Note:** In the lists below, '\*' indicates user input requested.

#### **List Commands**

@@CLR_LST	Clear/define a work list
@@MAK_LSTS	Make a list of selected fields from another list(s)*
@@MRG_LSTS	Merge a list(s) with another list(s)

#### **File Commands**

@@GET_FILS	Get the files to be used in this template*
@@RTV_FLDS	Retrieve the fields in a file
@@RTV_KEYS	Retrieve the keys of a file
@@RTV_RELN	Retrieve the relationship between two files

#### **Control Command**

@@GOTO	Pass control to a label
@@IF	Test a condition and pass control to a label
@@LABEL	A label which is the subject of another command
@@CMP_IDX	Compare an index value
@@DEC_IDX	Decrement an index value
@@INC_IDX	Increment an index value
@@SET_IDX	Set an index value

#### **Other Commands**

@@COMMENT Add a comment line to the generated RDML code

@@QUESTION Ask a question and receive a reply\*

#### Also See

Template Commands and Variables in the *Technical Reference Guide* 10.2 Creating Application Templates

### **10.2.3 Template Special Variables**

Application templates may contain special variables. Special variables are predefined template variables whose values are assigned by LANSA or by template commands. These variables are used by template commands and can be substituted into RDML when the template executes. In this respect, special variables are similar to a LANSA system variable where the value of the variable is set by executing a program (a template command).

Special variables are used in both application templates commands and RDML commands. For example:

@@QUESTION PROMPT('Actual test condition...')

ANSWER(@@CANS002) HELPIDS(HELP010)

IF COND('@@CANS002')

#### ENDIF

The user is requested to enter a description of the test condition to be used in an RDML command. The user's reply is stored in a special variable @@CANS002. The value of @@CANS002 will be substituted into the RDML IF command.

There are four basic groups of special variables:

- general variables
- question and answer variables
- file variables
- list variables.

Special variables play a very important role in templates as they provide a means of accessing and manipulating information from the template commands.

#### Also See

Template Commands and Variables in the Technical Reference Guide

10.2 Creating Application Templates

### **10.3 Application Template Program Examples**

These examples are provided as examples **only** of application templates. You should read and understand these example before attempting to write/modify application templates yourself.

10.3.1 Data Entry Program

- 10.3.2 Header/Detail Style Inquiry Template
- 10. Application Templates

### 10.3.1 Data Entry Program

The following is an example of a **simple** application template for a data entry program:

/\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/ /\* ASK FOR THE "WORD" - ask the user a question. \*/ /\* Note the prompt text and extended prompt text that \*/ /\* can be entered on the command. More detailed help \*/ /\* can also be entered in HELP panels for the template. \*/ /\* Note the special format of the ANSWER parameter. \*/ /\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/ @@QUESTION PROMPT('Supply word to describe+ WHAT this data entry program wo+ rks with') ANSWER(@@CANS001) EX+ TEND('The word you specify here+ is used to build messages that + appear on the' 'data entry scre+ en panel. You should use ONE wo+ rd only, use lowercase' 'charac+ ters only and only use singular+ form (eg: "customer", "employee+ "" ""order"). Do NOT use more t+ han 18 characters in your answe+ r.' 'Use the HELP function key + for more information and exampl+ es.') HELPIDS (HELP010 HELP020 + HELP030 HELP040 HELP050) /\* \_\*/ /\* GET NAME OF JUST ONE PHYSICAL FILE \*/ /\* Ask the user to enter the name of a single primary \*/ /\* file used for data entry. Note the prompt text /\* and extended prompt text that can be entered on the \*/ /\* command. More detailed help can also be entered on \*/ \*/ /\* HELP panels for the template. @@GET\_FILS TO(1) PROMPT('Enter name of PHYSICAL file to be used by this template') E+ XTEND('The file name may be specifie+

d partially (to cause a partial' 'li+ st of available files to be displaye+ d), or in left blank (to cause a ful+ l list' 'of available files to be di+ splayed). When a list of files is d+ isplayed,' 'the file required may be+ selected from the list. ' ' 'Use t+ he HELP function key for more detail+ s about this template and' 'examples+ of the type of RDML applications it + can create.') HELPIDS(HELP010 HELP02+ 0 HELP030 HELP040 HELP050) /* ====================================
/* GET FIELDS OF CHOSEN FILE INTO LIST 1 */ /* Note that all lists must be defined by an */ /* @@CLR_LST command before being used in an */ /* application template. */
/* ====================================
/* GET KEYS OF CHOSEN FILE INTO LIST 2 */ /* =================================
<pre>@@RTV_KEYS OF_FILE(1) INTO_LST(2) /* ====================================</pre>
<pre>/* AND PUT RESULTS INTO LIST 3 */ /* Note how the keys of the file are used as a */ /* force list to ensure all the fields are chosen. */ /* Note the column headings for the selection column */ /* and that sequence numbers are pre-filled on all */ /* fields in the selection list. This allows fields */ /* to be ordered in the desired sequence. */ /* =================================</pre>
@@CLR_LST_NUMBER(3) @@MAK_LSTS FROM_LSTS(1) FORCE_LSTS(2) INTO_LSTS+ ((3 'Fields to' 'Appear on' 'Entry P+ anel' *SEQUENCE *ALL)) HELPIDS(HELP0+

10 HELP020 HELP030 HELP040 HELP050) /* ====================================
/* GET USER TO CHOOSE FIELDS TO WORK WITHIN */ /* PROGRAM AND PUT RESULTS INTO LIST 4 */ /* Note that this list only requires the user to */ /* enter anynon-blank character to select a field.*/ /* ==================================
<pre>@@CLR_LST NUMBER(4) @@MAK_LSTS FROM_LSTS(1) INTO_LSTS((4 'Fields to ' '+ Work with "in Program ' *YESNO *NO)) H+ ELPIDS(HELP010 HELP020 HELP030 HELP040 H+ ELP050) /* ====================================</pre>
<pre>/* MERGE FIELDS IN LIST 4 INTO LIST 3 AS *HIDDEN */ /* The fields selected as fields to be worked with in */ /* the program are merged to list 3 with the *HIDDEN */ /* attribute if not already selected in list 3 by the */ /* previous @@MAK_LSTS command. */ /* =================================</pre>
@@MRG_LSTS FROM_LSTS((4 *HIDDEN)) INTO_LST(3) /* ====================================
/* ASK HOW THE PANEL IS TO BE DESIGNED */ /* =================================
@@QUESTION PROMPT('Design fields on data entry+ panel DOWN the screen or ACROSS the+ screen') ANSWER(@@CANS002) EXTEND('+ Reply DOWN or ACROSS only.' 'If you+ r data entry panel contains 17 (or + less) fields, DOWN is the 'reco+ mmended value. If your data entry p+ anel contains more than 17' 'fields+ , ACROSS is the recommended value.'+ 'Use the HELP function key for more+ information and examples.') LOWER(*+ NO) VALUES(DOWN ACROSS) HELPIDS(HEL+ P010 HELP020 HELP030 HELP040 HELP05+ 0) /* ====================================
· · · · · · · · · · · · · · · · · · ·

/\* GENERATE THE RDML PROGRAM

/\* The following code consists only of RDML that \*/ \*/ /\* will appear in the generated RDML program. /\* Note the use of special @@ variables in the RDML \*/ /\* commands--these are substituted when the template \*/ /\* is executed. \*/ @@COMMENT 'Function control options' FUNCTION OPTIONS(\*NOMESSAGES \*DEFERWRITE) @@COMMENT 'Group and field declarations' /\* \*/ /\* The following command will not be accepted by the \*/ /\* CL syntax checker, it must be **forced** to be accepted \*/ /\* as it will be quite valid when the special variable \*/ /\* @@LST03 is relaced by the list 3 elements when \*/ /\* executing this template. Do not code \*/ /\* FIELDS((#@@LST03))as this will generate \*/ \*/ /\* an invalid RDML command. This error /\* may also be true for other RDML commands. It \*/ /\* will be necessary to force these errors to \*/ /\* be accepted also (e.g. the DESIGN parameter of \*/ \*/ /\* the REQUEST command) /\* \*/ GROUP\_BY NAME(#PANELDATA) FIELDS(@@LST03) @@COMMENT 'Issue initial data entry message' MSGID(DCU0010) MSGF(DC@M01) + MESSAGE MSGDTA(""@@CANS001"") @@COMMENT 'Do data entry until terminated by + EXIT or CANCEL' **BEGIN LOOP** @@COMMENT 'Request user inputs or corrects details' REQUEST FIELDS(#PANELDATA) DESIGN(\*@@CANS002)+ IDENTIFY(\*LABEL) @@COMMENT 'Perform any program level validation here' BEGINCHECK **ENDCHECK** @@COMMENT 'Attempt to insert data into the data base' FIELDS((#PANELDATA)) TO FILE(@@FNAME01) INSERT @@COMMENT 'If okay, reset fields and issue accepted + message'

#### FIELD(#PANELDATA) TO(\*DEFAULT) CHANGE MSGID(DCU0011) MSGF(DC@M01) + MESSAGE MSGDTA(""@@CANS001"") END LOOP /\* \_\_\_\_\_\_ \*/ /\* CLEAR ALL LISTS USED \*/ /\* At the end of all application templates, it \*/ /\* is suggested that all work lists in the \*/ /\* template are cleared to delete all work records \*/ /\* from the database. \*/ /\* \_\_\_\_\_\_ /\* \_\_\_\_\_ \*/ @@CLR\_LST NUMBER(1) LST NUMBER(2) @@CLR\_LST NUMBER(3) @@CLR\_LST NUMBER(4)

10.3 Application Template Program Examples

## **10.3.2 Header/Detail Style Inquiry Template**

This is an example of a **fairly complex** application template for a header/detail style inquiry program:

<pre>/* ====================================</pre>
<pre>@@GET_FILS TO(50) PHY_ONLY(*NO) SGL_ONLY(*NO)+ PROMPT('Enter the name of the base+ file to be used by this template')+ EXTEND('The file name may be speci+ fied partially (to cause a partia+ l' 'list of available files to be + displayed), or in left blank (to c+ ause a full list' 'of available fi+ les to be displayed). When a list+ of files is displayed,' 'the file + required may be selected from the + list.''' 'Use the HELP function + key for more details about this te+ mplate and' 'examples of the type+ of RDML applications it can create+ .') HELPIDS(HELP010 HELP020 HELP03+ 0 HELP040) /* ====================================</pre>
/* LOAD DETAILS OF FIELDS OF "HEADER" INTO LIST 1 */ /* LOAD DETAILS OF FIELDS OF "BROWSE" INTO LIST 2 */ /* Use special variable @@FAREAnn to separate fields in */ /* the header and browse portions of the panel. */ /* Note the use of an index to control the loading of */ /* multiple file information. */ /* ====================================
@@CLR_LST NUMBER(1) @@CLR_LST NUMBER(2) @@SET_IDX IDX_NAME(CF) TO(1) A10: @@LABEL

```
@@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) IF_GT(A20)
         COND((*IF @@FAREACF *NE B)) GOTO(A12)
  @@IF
  @@RTV FLDS FROM FILE(CF) INTO LST(2)
  @@GOTO
          LABEL(A14)
A12: @@RTV FLDS FROM FILE(CF) INTO LST(1)
A14: @@INC_IDX IDX_NAME(CF)
  @@GOTO LABEL(A10)
A20: @@LABEL
/* ASK THE USER TO SELECT THE HEADER FIELDS REQUIRED */
@@CLR_LST NUMBER(11)
@@MAK_LSTS FROM_LSTS(1) INTO_LSTS((11 'Fields in'+
    'Header' 'Area' *SEQUENCE *ALL)) HELPI+
    DS(HELP010 HELP020 HELP030 HELP040)
/* ______________________*/
/* ASK THE USER TO SELECT THE BROWSE FIELDS REQUIRED */
@@CLR_LST_NUMBER(22)
@@MAK_LSTS FROM_LSTS(2) INTO_LSTS((22 'Fields in'+
    'Detail/List' 'Area' *SEQUENCE *ALL)) +
    HELPIDS(HELP010 HELP020 HELP030 HELP0 +
    40)
/* ______________________*/
/* ASK THE USER HOW TO DESIGN THE PANELS
                                        */
/* _______________________*/
@@QUESTION PROMPT('Design fields in the header a +
    rea DOWN the screen or ACROSS the scre+
    en') ANSWER(@@CANS002) EXTEND('Reply D+
    OWN or ACROSS only.' 'If your header a+
    rea contains 10 (or less) fields, DOWN+
    is the 'recommended value.+
    If your header area contains more than+
    10' 'fields, ACROSS is the recommended+
    value.' 'Use the HELP function key for+
    more information and examples.') LOWER+
    (*NO) VALUES(DOWN ACROSS) HELPIDS(HELP+
    010 HELP020 HELP030 HELP040)
/* ______
```

```
/* MERGE ALL RELATED KEY FIELDS INTO LIST 11 OR LIST 22
                                           */
/* AS *HIDDEN FIELDS. LIST 3 IS A WORKING LIST ONLY
                                         */
/* Note the use of @@RTV_RELN command to get the keys of */
                          */
/* the secondary files.
@@SET_IDX IDX_NAME(CF) TO(2)
A30: @@LABEL
  @@CMP_IDX IDX_NAME(CF) IDX_VALUE(@@TFMX) +
      IF GT(A40)
  @@CLR_LST NUMBER(3)
  @@RTV_RELN OF_FILE(CF) INTO_LST(3)
  @@IF COND((*IF @@FAREACF *NE B)) +
      GOTO(A34)
        COND((*IF @@FRELRCF *EQ M)) +
  @@IF
      GOTO(A34)
  @@MRG_LSTS FROM_LSTS((3 *HIDDEN)) INTO_LST(22)
  @@GOTO LABEL(A36)
A34: @@MRG_LSTS FROM_LSTS((3 *HIDDEN)) INTO_LST(11)
A36: @@INC_IDX IDX_NAME(CF)
  @@GOTO LABEL(A30)
A40: @@LABEL
/* GENERATION OF RDML CODE STARTS HERE
                                       */
/* ______
                               _____
  FUNCTION OPTIONS(*NOMESSAGES *DEFERWRITE)
  GROUP BY NAME(#HEADER) FIELDS(@@LST11)
  DEF LIST NAME(#LIST)
      FIELDS((#LISTDUMMY *HIDDEN) @@LST22)
  @@COMMENT 'Loop until user EXITs or CANCELs'
  BEGIN LOOP
=============
/* REQUEST KEYS OF THE BASE FILE BE INPUT AND GET DATA
                                            */
@@CLR LST NUMBER(3)
  @@RTV_KEYS OF_FILE(1) INTO_LST(3)
R10: REQUEST FIELDS(@@LST03) DESIGN(*@@CANS002) +
      IDENTIFY(*LABEL)
/* GENERATE FETCH TO THE PRIMARY FILE
                                     */
```

```
/* _____
                    ______
  @@COMMENT COMMENT('Fetch file @@FNAME01 details
                                          ')
        FIELDS((#HEADER)) +
  FETCH
      FROM FILE(@@FNAME01) +
      WITH_KEY(@@LST03) NOT_FOUND(R10) +
      ISSUE MSG(*YES)
_____
/* GENERATE FETCHES TO ALL FILES IN THE HEADER AREA
                                            */
@@SET_IDX IDX_NAME(CF) TO(2)
H10: @@LABEL
  @@CMP_IDX_IDX_NAME(CF) IDX_VALUE(@@TFMX) +
      IF_GT(H20)
        COND((*IF @@FAREACF *EQ B)) GOTO(H15)
  @@IF
  @@CLR LST NUMBER(3)
  @@RTV RELN OF FILE(CF) INTO LST(3)
  @@COMMENT COMMENT('Fetch file @@FNAMECF details
                                          ')
        FIELDS((#HEADER)) FROM_FILE(@@FNAMECF) +
  FETCH
      WITH_KEY(@@LST03)
H15: @@INC_IDX IDX_NAME(CF)
  @@GOTO LABEL(H10)
H20: @@LABEL
/* NOW EXTRACT DATA TO BE PLACED INTO THE BROWSE LIST
@@SET IDX IDX NAME(CF) TO(2)
  @@SET IDX IDX NAME(SC) TO(0)
A50: @@LABEL
  @@CMP IDX IDX NAME(CF) IDX VALUE(@@TFMX) +
      IF GT(A80)
        COND((*IF @@FAREACF *NE B)) GOTO(A78)
  @@IF
  @@CLR LST NUMBER(3)
  @@RTV RELN OF FILE(CF) INTO LST(3)
        COND((*IF @@FRELRCF *EQ M)) GOTO(A55)
  @@IF
  /* FETCH INTO THE LIST ENTRY
                                   */
  @@COMMENT COMMENT('Fetch file @@FNAMECF details
                                          ')
        FIELDS((#LIST)) FROM FILE(@@FNAMECF) +
  FETCH
      WITH KEY(@@LST03)
  @@GOTO LABEL(A78)
```

```
/* THE ONE AND ONLY SELECT COMMAND
                                     */
A55: @@COMMENT COMMENT('Select all file @@FNAMECF details')
  @@INC_IDX IDX_NAME(SC)
  SELECT FIELDS((#LIST)) FROM FILE(@@FNAMECF) +
     WITH_KEY(@@LST03)
  @@GOTO LABEL(A78)
 /* INC INDEX AND LOOP AROUND
                                  */
A78: @@INC_IDX IDX_NAME(CF)
  @@GOTO LABEL(A50)
A80: @@LABEL
/* ______
/* ADD ENTRY AND ENDSELECT FOR THE LIST (IF SELECT USED)
@@CMP_IDX IDX_NAME(SC) IDX_VALUE(0) IF_EQ(A90)
  ADD ENTRY TO LIST(#LIST)
  ENDSELECT
A90: @@LABEL
/* DISPLAY DETAILS TO THE USER
                               */
/* ______
  @@COMMENT COMMENT('Display results to the user')
 DISPLAY FIELDS(#HEADER) DESIGN(*@@CANS002)+
     IDENTIFY(*LABEL)+
     BROWSELIST(#LIST)
  @@COMMENT COMMENT('Clear header and list and +
     loop around ')
  CHANGE
         FIELD(#HEADER) TO(*DEFAULT)
  @@CMP IDX IDX NAME(SC) IDX VALUE(0) IF EQ(A95)
  CLR LIST NAMED(#LIST)
A95: @@LABEL
  END LOOP
/* CLEAR ALL LISTS USED
                            */
/* ______
  @@CLR LST NUMBER(1)
  @@CLR LST NUMBER(2)
  @@CLR LST NUMBER(3)
  @@CLR LST NUMBER(11)
  @@CLR LST NUMBER(12)
```

# 10.3 Application Template Program Examples

### **Appendix A. System Variable Evaluation Programs**

A system variable is a name assigned to a commonly used (and often variable) piece of information. For example, the current date is a commonly used piece of information that varies daily. It could be assigned the system variable name \*TODAYS\_DATE.

When a system variable is defined to LANSA, a name, description, method of derivation, type and length must be specified.

In addition, the name of a LANSA function or 3GL program that will "evaluate" the system variable must also be specified. It is this function/ program that is important because it provides the "intelligence" required to decide what is the correct value to place in the system variable.

When a system variable evaluation function/program is called by LANSA it is always passed 2 parameters. These are:

#### Number Type Length Description

1	А	20	System variable symbolic name
2	varies	Varies	Return storage for evaluated system variable.

The evaluation program examples supplied in this appendix are:

- RDML Functions
- 3GL Programs

### **RDML Functions**

Consider the following function to evaluate the system variable \*SYDN as an example of the simplest form of a system evaluation function:

FUNCTION OPTIONS(\*DIRECT \*ALP\_SYSTEM\_VARIABLE \*NOMESS CHANGE FIELD(#SYSVAR\$AV) TO('SYD0386')

• This function can be very easily expanded to return the values for 2 system variables named \*SYDN and \*MELB because they are both of type alphanumeric:

```
FUNCTION OPTIONS(*DIRECT *ALP_SYSTEM_VARIABLE *NOMESS
IF COND(#SYSVAR$NM *EQ '''*SYDN''')
CHANGE FIELD(#SYSVAR$AV) TO('SYD0386')
ELSE
CHANGE FIELD(#SYSVAR$AV) TO('MEB2307')
ENDIF
```

• In addition this function could return values for system variables that have different lengths like \*SYDN, \*MELB, \*COMPANY ...

FUNCTION OPTIONS(\*DIRECT \*ALP\_SYSTEM\_VARIABLE \*NOMESS CASE OF\_FIELD(#SYSVAR\$NM) WHEN VALUE\_IS('= "\*SYDN'") CHANGE FIELD(#SYSVAR\$AV) TO('SYD0386') WHEN VALUE\_IS('= "\*MELB'") CHANGE FIELD(#SYSVAR\$AV) TO('MEB2307') WHEN VALUE\_IS('= "\*COMPANY'") CHANGE FIELD(#SYSVAR\$AV) TO('C05') ENDCASE

In these examples the returned values are generated from moves of numeric or alphanumeric literals that are "hard-coded" into the system variable evaluation function. In reality this is not usually the case. The returned values are most often derived from data areas or database files. The extension of the logic demonstrated above to access data areas or database files is fairly easy to imagine.

### **Technical Notes**

System variable evaluation functions can handle system variables of different lengths and decimal precision but not of different types. The option \*ALP\_SYSTEM\_VARIABLE is used to indicate that this is a function to evaluate an alphanumeric value. The option \*NUM\_SYSTEM\_VARIABLE is used to indicate that this is a function to evaluate a numeric value.

It is necessary to define the following fields in the data dictionary, in order to access the system variable name and the system variable value within the evaluation function:

SYSVAR\$NM A(20)	System Variable Name
SYSVAR\$AV A(256)	System Variable Alphanumeric Return Value
SYSVAR\$NV P(30,9)	System Variable Numeric Return Value

Note that this implementation effectively prohibits evaluating numeric fields with more than 21 significant digits of precision.

If option \*ALP\_SYSTEM\_VARIABLE is used within the function, return the evaluated value in field SYSVAR\$AV, otherwise return the value in field SYSVAR\$NV if option \*NUM\_SYSTEM\_VARIABLE has been used. If either \*ALP\_SYSTEM\_VARIABLE or \*NUM\_SYSTEM\_VARIABLE have been entered as a function option, the following **design** constraints (rather than technical constraints) exist to ensure the correct use of the system variable facility:

- No DISPLAY, REQUEST or POP\_UP command can be used.
- No CALL can exist to another process/function within a complex logic validation function. However, a call to a 3GL program can exist.
- System variable evaluation functions cannot exist within an action bar process. This is not to say that they cannot be referenced by an action bar process, just not defined as part of a process of action bar type.
- System variable evaluation functions cannot have options of RCV\_DS or RCV\_LIST.
- The associated process must not have parameters.
- The exchange list may not be used. This restriction ensures insulated modularity in the evaluation function. System variables can be referenced

almost anywhere in LANSA and requiring information in an exchange list at the time of reference would require very strange implementations.

• Recursive implementations may be defined, but will fail to execute correctly. For instance: system variable \*TEST is defined and its evaluation function FUNC01 defined. If FUNC01 makes a reference to \*TEST directly, or indirectly (through a validation rule or default value) it will attempt to invoke itself recursively and fail.

The trap to watch out for here is in default values. Say field #TEST is defined in the data dictionary with a default of \*TEST. If FUNC01 makes any reference at all to field #TEST then it will be implemented with initialization logic to set up the default value of #TEST, which means of course a call to function FUNC01, which is not only recursive, but endless.

- Use of options \*DBOPTIMIZE and \*NOMESSAGES are recommended for system variable evaluation functions. The use of \*HEAVYUSAGE may also be considered in heavily used evaluation functions.
- The use of option \*MLOPTIMIZE is **strongly** recommended in all multilingual applications of this facility.

### **3GL Programs**

This program can be very easily expanded to return the values for the 2 system variables named \*SYDN and \*MELB because they have the same type and length:

С	*ENTRY PLIS	T
С	PARM	SYSVAR 20
С	PARM	VALUE 7
C*		
С	SYSVAR IFEC	) '*SYDN'
С	MOVEL'S	YD0386' VALUE
С	ELSE	
С	MOVEL'N	IEB2307' VALUE
С	END	
C*		
С	RETRN	

In addition this program could be modified to return values for system variables that have different types and lengths like \*SYDN, \*MELB, \*PERIOD and \*COMPANY:

IVALUI	E DS		
Ι	1	7 OFFICE	
Ι	Р	1 30PERIOD	
Ι	1	2 COMPNY	
C*			
С	*ENTRY PLIST	1	
С	PARM	SYSVAR 20	
С	PARM VALUE		
C*			
С	SYSVAR IFEQ	'*SYDN'	
С	MOVEL'SY	D0386' OFFICE	
С	ELSE		
С	SYSVAR IFEQ	'*MELB'	
С	MOVEL'MI	EB2307' OFFICE	
С	ELSE		
С	SYSVAR IFEQ	'*PERIOD'	
С	Z-ADD8701	1 PERIOD	
С	ELSE		

С	SYSVAR	IFEQ '*	*COMPANY'
С	MOV	VEL'05'	COMPNY
С	END	)	
C*			
С	RET	RN	

In addition, this program could be modified to return values for system variables that have different types and lengths such as \*SYDN, \*MELB, \*PERIOD and \*COMPANY:

In all cases the returned values are generated from moves of numeric or alphanumeric literals that are "hard-coded" into the system variable evaluation program. In reality this is not usually the case. The returned values are most often derived from data areas or database files. The extension of the logic demonstrated above to access data areas or database files is fairly easy to imagine.

### **Smart System Variables**

Some system variable evaluation programs are provided with the shipped version of LANSA. These are:

M@SYSDTA - Retrieve portion of an alphanumeric data area

M@SYSNUM- Get the next number from a data area

M@SYSDTA can be used to retrieve data from predefined alphanumeric data areas on your system.

M@SYSNUM can be used to generate the next number from a value that is stored in a data area. This can be returned as either an alpha or numeric value.

These shipped programs can be used as the system variable evaluation program for user defined system variables that follow a defined naming convention. This convention is shown below.

### Example 1

For **M@SYSDTA** use the following naming convention:

\*DTAssslllxxxxxxxxx

where

Sss Starting position in data area

Ill The length to retrieve (limited to 256 characters)

Xxxxxxxxx The data area name

#### Note:

- The data area must exist in the library list.
- Data area name cannot be \*LDA (Local Data Area) or \*GDA (Group Job Data Area).
- Method of derivation should be STATIC for system variables using this evaluation program.

#### Example of Use (#1)

Retrieve the company name from data area COMPDETL. The company is in position 32 and is 25 characters long.

#### **STEP 1 - Define System Variable**

#### STEP 2 - Use in a LANSA Function

#### Example 2

For **M@SYSNUM** use the following naming conventions . . .

For alphanumeric results \*AUTOALPnnxxxxxxxx

For numeric results \*AUTONUMnnxxxxxxxx

where nn = is length to retrieve and xxxxxxxx = the data area name.

#### Note:

- Data area name cannot be \*LDA (Local Data Area) or \*GDA (Group Job Data Area).
- If data area cannot be allocated / found, it will be created with name xxxxxxxx of type \*DEC and length nn (limited to 15 digits), and initialized to 0.
- The xxxxxxxx part of system variable name must be unique.
- Method of derivation should be DYNAMIC for system variables using this evaluation program.

#### Example of Use (#2)

Retrieve the next available order number. Used order numbers are stored in a data area called ORDNO. This data area is defined as decimal 5 with 0 decimal places. The order number should be returned as alphanumeric with length 5.

#### **STEP 1 - Define System Variable**

#### STEP 2 - Use in a LANSA Function

### **Appendix B. Domino Integration Examples**

These Domino Integration examples are provided: Example 1: Creating a New Document in a Database Example 2: Selecting Documents from a Database using a View Example 3: Executing an Agent in a Database Example 4: Updating Documents Selected from a Browse List Before attempting to use these examples you should:

- Use the Lotus Notes client to create a database named BIFTEST with text field INVCODE, numeric field INVAMT, Date/Time field of INVDATE.
- Create a view to display the fields of BIFTEST.
- Create an agent that changes database data.

#### **Example 1: Creating a New Document in a Database**

```
********* Beginning of RDML commands *********
FUNCTION OPTIONS(*DIRECT)
        FIELD(#DATABASE) TYPE(*CHAR) LENGTH(50)
DEFINE
     INPUT ATR(LC)
     DEFAULT(""BIFTEST"")
        FIELD(#SERVER) TYPE(*CHAR) LENGTH(50) INPUT ATF
DEFINE
        FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)
DEFINE
        FIELD(#DBHANDLE) TYPE(*CHAR) LENGTH(4)
DEFINE
DEFINE
        FIELD(#DOCHANDLE) TYPE(*CHAR) LENGTH(4)
        FIELD(#ITEMNAME) TYPE(*CHAR) LENGTH(20)
DEFINE
        FIELD(#ITEMTYPE) TYPE(*DEC) LENGTH(1) DECIMALS
DEFINE
        FIELD(#INVCODE) TYPE(*CHAR) LENGTH(10)
DEFINE
        FIELD(#INVAMT) TYPE(*DEC) LENGTH(15) DECIMALS(2
DEFINE
     EDIT CODE(3)
DEFINE
        FIELD(#INVDATE) TYPE(*CHAR) LENGTH(20)
     INPUT ATR(LC)
        FIELD(#DUMMY) TYPE(*CHAR) LENGTH(1)
DEFINE
REQUEST FIELDS((#DATABASE *L5 *P2) (#SERVER *L9 *P2))
     DESIGN(*DOWN) IDENTIFY(*LABEL) DOWN_SEP(001)
     ACROSS_SEP(001)
******** Open the database
USE
       BUILTIN(DOM OPEN DATABASE) WITH ARGS(#DATABASE)
     #DUMMY #SERVER) TO GET(#RETCODE #DBHANDLE)
IF
     COND('#RETCODE *NE "OK"')
MENU
ENDIF
****** Add new documents to database
BEGIN LOOP
REQUEST FIELDS((#INVCODE *L3 *P2) (#INVAMT *L5 *P2)
     (#INVDATE *L7 *P2)) DESIGN(*DOWN) IDENTIFY(*LABEL)
     DOWN SEP(001) ACROSS SEP(001) EXIT KEY(*YES L1)
     MENU KEY(*YES L1)
******** Create a new document in the database
       BUILTIN(DOM CREATE DOCUMENT) WITH ARGS(#DBH/
USE
     TO GET(#RETCODE #DOCHANDLE)
IF
     COND('#RETCODE *NE "OK"')
```

GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*\* Add data items to the document

CHANGE FIELD(#ITEMNAME) TO(INVCODE)

CHANGE FIELD(#ITEMTYPE) TO(2)

USE BUILTIN(DOM\_ADD\_ITEM) WITH\_ARGS(#DOCHANDLE #ITEMNAME #ITEMTYPE #INVCODE)

TO\_GET(#RETCODE)

- IF COND('#RETCODE \*NE "OK"')
- GOTO LABEL(L1)

ENDIF

CHANGE FIELD(#ITEMNAME) TO(INVAMT)

CHANGE FIELD(#ITEMTYPE) TO(1)

USE BUILTIN(DOM\_ADD\_ITEM) WITH\_ARGS(#DOCHANDLE #ITEMNAME #ITEMTYPE #DUMMY #INVAMT) TO GET(#RETCODE)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

CHANGE FIELD(#ITEMNAME) TO(INVDATE)

CHANGE FIELD(#ITEMTYPE) TO(3)

USE BUILTIN(DOM\_ADD\_ITEM) WITH\_ARGS(#DOCHANDLE #ITEMNAME #ITEMTYPE #INVDATE) TO\_GET(#RETCODE)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*\*\*\* Update the new document in the database

USE BUILTIN(DOM\_UPDATE\_DOCUMENT) WITH\_ARGS(#DOCI TO\_GET(#RETCODE)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ELSE

MESSAGE MSGTXT('New document added to BIF Test Database') ENDIF

\*\*\*\*\*\*\*\*\*\*\* Close the document to release memory

USE BUILTIN(DOM\_CLOSE\_DOCUMENT) WITH\_ARGS(#DOCH TO\_GET(#RETCODE)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

#### ENDIF END\_LOOP \*\*\*\*\*\*\*\*\* Close the database upon exit or an error L1 USE BUILTIN(DOM\_CLOSE\_DATABASE) WITH\_ARGS(#DBHA) TO\_GET(#RETCODE) \*\*\*\*\*\*\*\*\* End of RDML commands \*\*\*\*\*\*\*\*

#### **Example 2: Selecting Documents from a Database using a View**

```
********* Beginning of RDML commands *********
FUNCTION OPTIONS(*DIRECT)
******** Field definitions
        FIELD(#DATABASE) TYPE(*CHAR) LENGTH(50)
DEFINE
     INPUT ATR(LC) DEFAULT("BIFTEST")
DEFINE
        FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)
        FIELD(#DBHANDLE) TYPE(*CHAR) LENGTH(4)
DEFINE
        FIELD(#DOCHANDLE) TYPE(*CHAR) LENGTH(4)
DEFINE
        FIELD(#VIEW) TYPE(*CHAR) LENGTH(80) INPUT_ATR(L
DEFINE
        FIELD(#IDTHANDLE) TYPE(*CHAR) LENGTH(4)
DEFINE
        FIELD(#NUMNOTES) TYPE(*DEC) LENGTH(7) DECIMAL
DEFINE
DEFINE
        FIELD(#LSTNOTEID) TYPE(*DEC) LENGTH(7) DECIMAL:
        FIELD(#NOTEID) TYPE(*DEC) LENGTH(7) DECIMALS(0)
DEFINE
        FIELD(#ITEMNAME) TYPE(*CHAR) LENGTH(20)
DEFINE
DEFINE
        FIELD(#ITEMTYPE) TYPE(*DEC) LENGTH(1) DECIMALS
DEFINE
        FIELD(#INVCODE) TYPE(*CHAR) LENGTH(10)
        FIELD(#INVAMT) TYPE(*DEC) LENGTH(15) DECIMALS(2
DEFINE
     EDIT CODE(3)
        FIELD(#INVDATE) TYPE(*CHAR) LENGTH(20)
DEFINE
     INPUT ATR(LC)
DEFINE
        FIELD(#DUMMY) TYPE(*CHAR) LENGTH(1)
DEF LIST NAME(#LIST1) FIELDS((#INVCODE) (#INVAMT)
     (#INVDATE))
******** Open the database
       BUILTIN(DOM OPEN DATABASE) WITH ARGS(#DATABSE
USE
     TO GET(#RETCODE #DBHANDLE)
IF
     COND('#RETCODE *NE "OK"')
MENU
ENDIF
******
BEGIN LOOP
CLR_LIST NAMED(#LIST1)
REQUEST FIELDS((#VIEW *L7 *P2)) DESIGN(*DOWN)
     IDENTIFY(*LABEL) DOWN SEP(001) ACROSS SEP(001)
     EXIT KEY(*YES L1) MENU KEY(*YES L1)
```

```
TEXT((*TMAP001 1 1))
```

\*\*\*\*\*\*\*\*\* Search using the requested view

USE BUILTIN(DOM\_SEARCH\_DOCUMENTS) WITH\_ARGS(#DBI 'V' #VIEW) TO\_GET(#RETCODE #IDTHANDLE #NUMNOTES)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*\*\* Read all documents in Note ID Table

IF COND('#NUMNOTES > 0')

CHANGE FIELD(#LSTNOTEID) TO(0)

USE BUILTIN(DOM\_GET\_NXT\_DOCUMENT) WITH\_ARGS(#IDT #LSTNOTEID)

TO\_GET(#RETCODE #NOTEID)

\*\*\*\*\*\*\*

DOWHILE COND('#RETCODE \*NE "EF"')

CHANGE FIELD(#LSTNOTEID) TO(#NOTEID)

```
*****
```

```
USE BUILTIN(DOM_OPEN_DOCUMENT) WITH_ARGS(#DBHAN
#NOTEID) TO_GET(#RETCODE #DOCHANDLE)
```

\*\*\*\*\*\*\*\*\* Get data items from the document

CHANGE FIELD(#ITEMNAME) TO(INVCODE)

CHANGE FIELD(#ITEMTYPE) TO(2)

USE BUILTIN(DOM\_GET\_ITEM) WITH\_ARGS(#DOCHANDLE #ITEMNAME #ITEMTYPE) TO\_GET(#RETCODE #INVCODE)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

```
CHANGE FIELD(#ITEMNAME) TO(INVAMT)
```

```
CHANGE FIELD(#ITEMTYPE) TO(1)
```

USE BUILTIN(DOM\_GET\_ITEM) WITH\_ARGS(#DOCHANDLE #ITEMNAME #ITEMTYPE)

TO\_GET(#RETCODE #DUMMY #INVAMT)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*\*\*\*\* COND('#NUMNOTES \*EQ 0')

CHANGE FIELD(#ITEMNAME) TO(INVDATE)

CHANGE FIELD(#ITEMTYPE) TO(3)

USE BUILTIN(DOM\_GET\_ITEM) WITH\_ARGS(#DOCHANDLE #ITEMNAME #ITEMTYPE) TO\_GET(#RETCODE #INVDATE) IF COND('#RETCODE \*NE "OK''')

GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*\*\* Add to list

ADD\_ENTRY TO\_LIST(#LIST1)

\*\*\*\*\*\*\*\*\*\*\* Close the document to release memory

USE BUILTIN(DOM\_CLOSE\_DOCUMENT) WITH\_ARGS(#DOCH/ TO\_GET(#RETCODE)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*

USE BUILTIN(DOM\_GET\_NXT\_DOCUMENT) WITH\_ARGS(#IDTHANDLE #LSTNOTEID) TO GET(#RETCODE #NOTEID)

ENDWHILE

\*\*\*\*\*

ENDIF

USE BUILTIN(DOM\_END\_SEARCH\_DOCS) WITH\_ARGS(#IDTH/ TO\_GET(#RETCODE)

\*\*\*\*\*\*\*\*\* Show selected documents in a list

DISPLAY FIELDS((#NUMNOTES)) BROWSELIST(#LIST1)

EXIT\_KEY(\*YES L1) MENU\_KEY(\*YES L1)

\*\*\*\*\*

END\_LOOP

\*\*\*\*\*\*\*\*\*\* Close the database upon exit or an error

L1 USE BUILTIN(DOM\_CLOSE\_DATABASE) WITH\_ARGS(#DBHA) TO\_GET(#RETCODE)

\*\*\*\*\*\*\*\*\*\* End of RDML commands \*\*\*\*\*\*\*\*\*

#### **Example 3: Executing an Agent in a Database**

```
********* Beginning of RDML commands *********
 FUNCTION OPTIONS(*DIRECT)
 ******** Field definitions
          FIELD(#DATABASE) TYPE(*CHAR) LENGTH(50)
 DEFINE
      INPUT ATR(LC) DEFAULT("BIFTEST")
          FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)
 DEFINE
          FIELD(#DBHANDLE) TYPE(*CHAR) LENGTH(4)
 DEFINE
 DEFINE
          FIELD(#AGENTNAME) TYPE(*CHAR) LENGTH(50)
       INPUT ATR(LC)
 ******** Open the database
         BUILTIN(DOM OPEN DATABASE) WITH ARGS(#DATABASE)
 USE
      TO GET(#RETCODE #DBHANDLE)
 ****** Execute nominated agent in database
 BEGIN LOOP
 REQUEST FIELDS((#AGENTNAME *L3 *P2)(#RETCODE *L6 *P2))
      DESIGN(*DOWN) IDENTIFY(*LABEL) DOWN SEP(001)
      ACROSS SEP(001) EXIT KEY(*YES L1) MENU KEY(*YES L1)
      TEXT((*TMAP001 1 1)
      )
         BUILTIN(DOM EXECUTE AGENT)
 USE
       WITH ARGS(#DBHANDLE #AGENTNAME) TO GET(#RETCO]
       COND('#RETCODE *NE "OK"')
 IF
 GOTO
          LABEL(L1)
 ELSE
 MESSAGE MSGTXT('Agent executed successfully')
 ENDIF
 END LOOP
 ********** Close the database upon exit or an error
          BUILTIN(DOM CLOSE DATABASE) WITH ARGS(#DBHA)
L1 USE
      TO GET(#RETCODE)
```

\*\*\*\*\*\*\*\*\* End of RDML commands \*\*\*\*\*\*\*\*

### **Example 4: Updating Documents Selected from a Browse List**

This example shows updating documents selected from a browse list after selection from a database using selection criteria.

selection from a database doing selection efferta.					
********** Beginning of RDML commands *********					
FUNCTION OPTIONS(*DIRECT)					
********* Field definitions					
DEFINE FIELD(#DATABASE) TYPE(*CHAR) LENGTH(50) INPUT_AT					
DEFAULT("'BIFTEST''')					
DEFINE FIELD(#RETCODE) TYPE(*CHAR) LENGTH(2)					
DEFINE FIELD(#DBHANDLE) TYPE(*CHAR) LENGTH(4)					
DEFINE FIELD(#DOCHANDLE) TYPE(*CHAR) LENGTH(4)					
DEFINE FIELD(#SELTYPE) TYPE(*CHAR) LENGTH(1) DEFAULT(N)					
DEFINE FIELD(#CRITERIA) TYPE(*CHAR) LENGTH(80) INPUT_ATR					
DEFINE FIELD(#IDTHANDLE) TYPE(*CHAR) LENGTH(4)					
DEFINE FIELD(#NUMNOTES) TYPE(*DEC) LENGTH(7) DECIMALS(					
DEFINE FIELD(#LSTNOTEID) TYPE(*DEC) LENGTH(7) DECIMALS((					
DEFINE FIELD(#NOTEID) TYPE(*DEC) LENGTH(7) DECIMALS(0)					
DEFINE FIELD(#ITEMNAME) TYPE(*CHAR) LENGTH(20)					
DEFINE FIELD(#ITEMTYPE) TYPE(*DEC) LENGTH(1) DECIMALS(0)					
DEFINE FIELD(#INVCODE) TYPE(*CHAR) LENGTH(10)					
DEFINE FIELD(#INVAMT) TYPE(*DEC) LENGTH(15) DECIMALS(2)					
EDIT_CODE(3)					
EFINE FIELD(#INVDATE) TYPE(*CHAR) LENGTH(20) INPUT_ATE					
DEFINE FIELD(#DUMMY) TYPE(*CHAR) LENGTH(1)					
DEF_LIST NAME(#LIST1) FIELDS((#SELECT *SELECT) (#INVCODE)					
(#INVAMT) (#INVDATE) (#NOTEID *HIDDEN))					
********* Open the database					
USE BUILTIN(DOM_OPEN_DATABASE) WITH_ARGS('#DATABASE					
TO_GET(#RETCODE #DBHANDLE)					
IF COND('#RETCODE *NE "OK"')					
MENU					
ENDIF					
*****					
BEGIN_LOOP					
CLR_LIST NAMED(#LIST1)					
REQUEST FIELDS((#CRITERIA *L9 *P2)(#SELTYPE *L5 *P2 *DESC))					

```
DESIGN(*DOWN) IDENTIFY(*LABEL) DOWN_SEP(001)
ACROSS_SEP(001) EXIT_KEY(*YES L1) MENU_KEY(*YES L1)
TEXT((*TMAP001 1 1))
```

\*\*\*\*\*\*\*\* Search all documents or using criteria

- USE BUILTIN(DOM\_SEARCH\_DOCUMENTS) WITH\_ARGS(#DBHA #SELTYPE #DUMMY #CRITERIA) TO\_GET(#RETCODE #IDTHAN #NUMNOTES)
- IF COND('#RETCODE \*NE "OK"')

```
GOTO LABEL(L1)
```

```
ENDIF
```

```
******** Read all documents in Note ID Table
```

```
IF COND('#NUMNOTES > 0')
```

```
CHANGE FIELD(#LSTNOTEID) TO(0)
```

USE BUILTIN(DOM\_GET\_NXT\_DOCUMENT) WITH\_ARGS(#IDTH/ #LSTNOTEID) TO\_GET(#RETCODE #NOTEID)

```
******
```

DOWHILE COND('#RETCODE \*NE "EF"')

```
CHANGE FIELD(#LSTNOTEID) TO(#NOTEID)
```

```
*****
```

```
USE BUILTIN(DOM_OPEN_DOCUMENT) WITH_ARGS(#DBHANDI
#NOTEID) TO_GET(#RETCODE #DOCHANDLE)
```

\*\*\*\*\*\*\*\*\* Get data items from the document

```
CHANGE FIELD(#ITEMNAME) TO(INVCODE)
```

```
CHANGE FIELD(#ITEMTYPE) TO(2)
```

```
USE BUILTIN(DOM_GET_ITEM) WITH_ARGS(#DOCHANDLE #ITE
#ITEMTYPE) TO_GET(#RETCODE #INVCODE)
```

```
IF COND('#RETCODE *NE "OK'")
```

```
GOTO LABEL(L1)
```

ENDIF

```
CHANGE FIELD(#ITEMNAME) TO(INVAMT)
```

```
CHANGE FIELD(#ITEMTYPE) TO(1)
```

```
USE BUILTIN(DOM_GET_ITEM) WITH_ARGS(#DOCHANDLE #ITE
#ITEMTYPE) TO_GET(#RETCODE #DUMMY #INVAMT)
```

```
IF COND('#RETCODE *NE "OK"')
```

```
GOTO LABEL(L1)
```

ENDIF

CHANGE FIELD(#ITEMNAME) TO(INVDATE)

```
CHANGE FIELD(#ITEMTYPE) TO(3)
```

USE BUILTIN(DOM\_GET\_ITEM) WITH\_ARGS(#DOCHANDLE #ITE

#ITEMTYPE) TO\_GET(#RETCODE #INVDATE)

IF COND('#RETCODE \*NE "OK'")

GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*\*\* Add to list

ADD\_ENTRY TO\_LIST(#LIST1)

\*\*\*\*\*\*\*\*\*\*\* Close the document to release memory

```
USE BUILTIN(DOM_CLOSE_DOCUMENT) WITH_ARGS(#DOCHAN
TO_GET(#RETCODE)
```

IF COND('#RETCODE \*NE "OK'")

```
GOTO LABEL(L1)
```

ENDIF

\*\*\*\*\*\*

```
USE BUILTIN(DOM_GET_NXT_DOCUMENT) WITH_ARGS(#IDTH/
#LSTNOTEID) TO_GET(#RETCODE #NOTEID)
```

ENDWHILE

\*\*\*\*\*

#### ENDIF

\*\*\*\*\*\*

```
USE BUILTIN(DOM_END_SEARCH_DOCS) WITH_ARGS(#IDTHAN TO_GET(#RETCODE)
```

**\*\*\*\*\*\*\*\*** Show selected documents in a list

DISPLAY FIELDS((#NUMNOTES)) BROWSELIST(#LIST1) EXIT\_KEY( L1) MENU\_KEY(\*YES L1)

**\*\*\*\*\*\*\*** Select documents from list to be updated

SELECTLIST NAMED(#LIST1) GET\_ENTRYS(\*SELECT)

\*\*\*\*\*\*\*\*\* Open the selected document

USE BUILTIN(DOM\_OPEN\_DOCUMENT) WITH\_ARGS(#DBHANDI #NOTEID) TO\_GET(#RETCODE #DOCHANDLE)

IF COND('#RETCODE \*NE "OK"')

```
GOTO LABEL(L1)
```

ENDIF

**\*\*\*\*\*\*\*** Update the invoice amount in selected documents

CHANGE FIELD(#INVAMT) TO('#INVAMT + 10')

CHANGE FIELD(#ITEMNAME) TO(INVAMT)

CHANGE FIELD(#ITEMTYPE) TO(1)

USE BUILTIN(DOM\_UPDATE\_ITEM) WITH\_ARGS(#DOCHANDLE #ITEMNAME #ITEMTYPE #DUMMY #INVAMT) TO\_GET(#RETC(

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

**\*\*\*\*\*\*** Update the selected document in the database

- USE BUILTIN(DOM\_UPDATE\_DOCUMENT) WITH\_ARGS(#DOCHA TO\_GET(#RETCODE)
- IF COND('#RETCODE \*NE "OK"')
- GOTO LABEL(L1)

ENDIF

\*\*\*\*\*\*\*\*\*\* Close the document to release memory

USE BUILTIN(DOM\_CLOSE\_DOCUMENT) WITH\_ARGS(#DOCHAN TO\_GET(#RETCODE)

IF COND('#RETCODE \*NE "OK"')

GOTO LABEL(L1)

ENDIF

ENDSELECT

\*\*\*\*\*\*

END\_LOOP

\*\*\*\*\*\*\*\*\*\* Close the database upon exit or an error

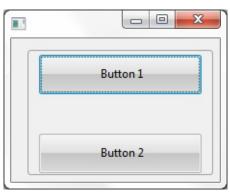
L1 USE BUILTIN(DOM\_CLOSE\_DATABASE) WITH\_ARGS(#DBHA) TO\_GET(#RETCODE)

\*\*\*\*\*\*\*\*\* End of RDML commands \*\*\*\*\*\*\*\*

### Set the Properties of the Child Components

When applicable you can change the properties of the child components in the layout using the child details tab.

This example form contains a group box with two buttons. The group box is attached to the form with an attachment layout and the buttons are attached to the group box with another attachment layout:

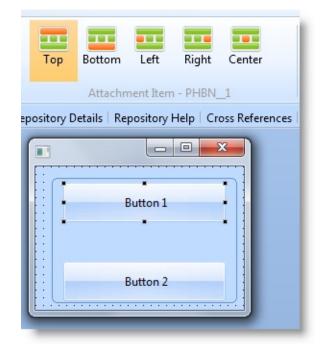


Here is the definition for the form:

```
Function Options(*DIRECT)
```

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Height(187) Layoutmanager(# Define\_Com Class(#PRIM\_GPBX) Name(#GPBX\_1) Displayposition(1) Hei Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) Caption('Button 1') Disj Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_2) Caption('Button 2') Disj Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1) Marginbottom(5) Marg Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Center) Mana Define\_Com Class(#PRIM\_ATLM) Name(#ATLI\_2) Marginleft(10) Margint Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_5) Attachment(Top) Manage Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_6) Attachment(Bottom) Man End\_Com

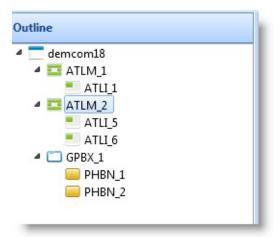
You can see in the ribbon that the first button is attached to the top of the layout and the second button is attached to the bottom of it:



↑ Set the Layout Properties

### **Display the Layout Manager**

Layout managers are visible in the Outline tab:



If you have the Show Layout Items option selected in LANSA settings, the layout items attached to all the components in the layoutmanaged component are displayed.

The layout items have their own properties, such as the name of the component it manages, the container, its layout manager, position, margins:

TLI_5	
Ē ▼ ?	
Properties Events Me	ethods
🖻 Attachment	Тор
ComponentClass	PRIM_ATLI
🖉 ComponentPatter	PRIM_ATLI
🖻 ComponentTag	
ComponentTypeN	PRIM_ATLI
🖻 Manage	#PHBN_1
🖉 MarginBottom	0
🖻 MarginLeft	0
P MarginRight	0
Margin Top	0
P MaxHeight	0
MaxWidth	0
🖻 MinHeight	0
MinWidth	0
P Name	ATLI_5
P Owner	#DEMCOM18
Parent	#ATLM_2

Normally you do not work with the individual layout items but instead define the entire layout using the Layout Helper.

To turn off the display of layout items in the Outline tab, deselect the Show Layout Items option in the LANSA Settings dialog accessed from the Options tab.

#### **The Layout Manager Definitions**

When you choose a layout, a layout manager DEFINE\_COM statement is automatically created in the source of your container:

Define\_Com Class(#PRIM\_FWLM) Name(#FWLM\_1) Direction(TopToBottc

The class and the name of the layout manager depend on its type:

Туре	Name
Attachment layout manager	#ATLM_n
Grid layout manager	#GDLM_n
Flow layout manager	#FWLM_n
Split layout manager	#SPLM_n

For every component added to a layout, a layout item is added to keep the component attached to a specified place in its parent container's layout. So when you add components to a container with a layout manager, a layout item definition that determines the position of the component is also added:

DEFINE\_COM CLASS(#PRIM\_PHBN) NAME(#PHBN\_1) CAPTION('Top') DEFINE\_COM CLASS(#PRIM\_ATLI) NAME(#ATLI\_1) ATTACHMENT(To

The layout items themselves are named in the same way as the layout managers:

Туре	Class and Name
Attachment layout manager item	#ATLI_n
Grid layout manger item	#GDLI_n
Flow layout manager item	#FWLI_n
Split layout manager item	#SPLI_n

# **Attachment Layout**

An attachment layout divides the container into five cells. The center cell expands to fill any available space.

When you add components on an attachment layout, the position where you drop them in the layout determines where they will be positioned. So components dropped to the top of a form with attachment layout will be assigned Attachment position Top, components dropped to bottom will be assigned Attachment position Bottom etc. You can change the position using the Layout Helper.

This form contains five push-buttons, each assigned to one of the five Attachment positions.

	Button 1	Ĩ
Button 4	Button 5	Button 2
	Button 3	

This is the source for the layout:

Function Options(\*DIRECT)

Begin\_Com Role(\*EXTENDS #PRIM\_FORM) Height(304) Layoutmanager(# Define\_Com Class(#PRIM\_ATLM) Name(#ATLM\_1)

Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_1) Caption('Button 1') Disj Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_1) Attachment(Top) Manage Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_2) Caption('Button 2') Disj Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_2) Attachment(Right) Manag Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_3) Caption('Button 3') Disj Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_3) Attachment(Bottom) Man Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_4) Caption('Button 4') Disj Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_4) Attachment(Left) Manage Define\_Com Class(#PRIM\_PHBN) Name(#PHBN\_5) Caption('Button 5') Disj Define\_Com Class(#PRIM\_ATLI) Name(#ATLI\_5) Attachment(Center) Mana End\_Com

This is what the layout looks like in the Layout Helper (to display it click on the button with ellipses in the LayoutManager property of the Details tab of the container):

Layout Helper			×
Layout Managed Component	🔁 Layout		<b>×</b> 10
	- ATLM_1		
Children 🕖 🗙 🕖 🗙	Layout Manager Deta	ails Child Detai	ls As Child Details
<pre>PHBN_1 PHBN_2 PHBN_3 PHBN_4</pre>	Category Attach	ment	*
PHBN_5	💿 Тор	🔘 Left	O Center
	O Bottom	Right	None

The component in the center is always changed to accommodate the space left over from the other components.

You will often place Panel components in the attachment positions and then add other components to the panels.

Note that if more than one component is given the same Attachment position, only the component with the lowest Display position will be displayed, and the other components will be hidden.

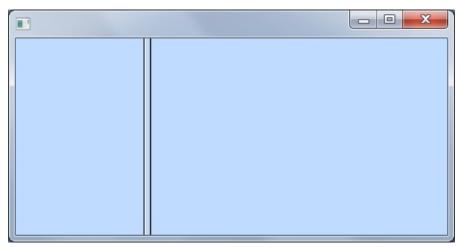
A very common use of Attachment layout is one where only the Top, Center and Bottom positions are used:

- The Top position contains a toolbar created from a panel and toolbar buttons
- The Centre position contains a panel with various fields
- The Bottom position contains a status bar

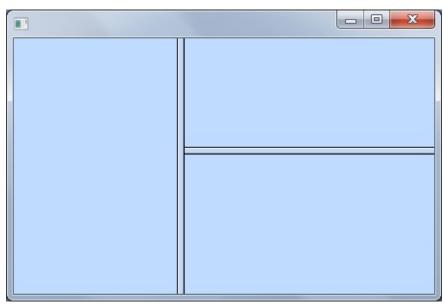
# Split Layout

When you specify a split layout manager (Split Vertically or Split Horizontally), two layout items and two panels are created automatically. These panels are separated by either a vertical or a horizontal sizing bar.

The layout items control the two panels which are separated by the sizing bar. As the sizing bar is moved with the mouse, the relative widths of the panels are adjusted:



Both panels themselves have a LayoutManager property that control the panel. Therefore you can, for example, split one of the panels using another split layout manager:



This is what the layout looks like in the Layout Helper (to display it click on the button with ellipses in the LayoutManager property of the Details tab of the

container):

Layout Helper	×
Layout Managed Component	€ Layout 😤 🗙
Children 🕖 🗙 🕖 🗙	Layout Manager Details Child Details As Child Details
PANL_1 PANL_2	Category Divider Style *
	Bar
	Gap
	<ul> <li>Raised</li> </ul>

Whether the layout is split horizontally or vertically can be changed using the Orientation property of the split layout manager.

The amount of space initially taken by the different panels is specified using the Weight property of the split layout items. If the Weight of a panel is 0, it will not be resized as the container is resized. You can fine tune the spaces taken by the panels by adjusting the relative proportions of the Weight property values of the panels.

## **Grid Layout**

In a grid layout you specify the Column from left and the Row from top where the components are placed:

Button 2	Button 5	
Button 1		
	Button 3	
Button 4		

The easiest way to position the components in the layout is using the Layout Helper (to display it click on the button with ellipses in the LayoutManager property of the Details tab of the container):

Layout Mana	gea C	omp	one	int		≚×
demcom18					- GDLM_1	
Children	Ű	×	U	×	Layout Manager Details Child Details As Child	Detail
V PHBN_1 V PHBN_2 V PHBN_3 V PHBN_4 V PHBN_5					Category Grid Position Row from Top 2 Column from Left 2 Kows 1 Columns 1 Colum	

### **Flow Layout**

The Direction property of the flow layout managers determines how components are placed on the container. The Direction can be:

- Top to Bottom or Bottom to Top. The items are arranged in columns and the columns are filled from either the top or the bottom.
- Left to Right or Right to Left. The items are arranged in rows and items are filled either form left or right.

Here is a form with a TopToBottom layout:

			×	
Button 5				
Button 4				
Button 3			· · · · · · · · · · · · · · · · · · ·	
Button 2	· · · · · · · · · · · · · · · · · · ·	 · · · · · · · · · · · · · · · · · · ·		
Button 1			· · · · · · · · · · · · · · · · · · ·	

And here is a LeftToRight layout:

Button 1	Button 2	Button 3	
Button 4	Button 5	Button 6	
Button 7			
			· · · · · · · · · · · · · · · · · · ·

When you add components to a container with a flow layout manager they are automatically aligned and adjusted in the specified order. The spacing between the components is automatic. The order in which the components are added is determined by their DisplayPosition property value.

This is what the layout looks like in the Layout Helper (to display it click on the button with ellipses in the LayoutManager property of the Details tab of the container):

Cayout Helper	×
Layout Managed Component	🔁 Layout 📑 🗙
demcom18	* FWLM_1 *
Children 🔋 🗙 🖞 🗙	Layout Manager Details Child Details As Child Details
<ul> <li>✓ PHBN_1</li> <li>✓ PHBN_2</li> <li>✓ PHBN_3</li> <li>✓ PHBN_4</li> <li>✓ PHBN_5</li> <li>✓ PHBN_6</li> <li>✓ PHBN_7</li> </ul>	Category Flow Rules

The flow layout managers have also various FlowOperation properties which can be used to fine tune how the rows and columns are filled. The ItemsPerDivision property can be used to specify the number of columns (Top to Bottom, Bottom to Top) or the number of rows (Left to Right, Right to Left). When this property is set to 0 the number of columns and rows will be adjusted automatically according to available space.

# **Reusable Part AXOBJECTA**

\* Reusable Part AXOBJECTA Function Options(\*DIRECT) Begin\_Com Role(\*EXTENDS #PRIM\_OBJT) Attribute Class(#PRIM\_ATTR.AX\_TYPELIB) Guid('{F86172A5-3FF2-4864-869A-8B1A435C65CA}') TypeLibName('LANSA\_AXOBJECTA\_LIB') Attribute Class(#PRIM\_ATTR.AX\_CLASS) Guid('{4FD69BDC-F4F5-4AB2-8869-9708CA6AE921}') ProgId('LANSA.AXOBJECTA') Attribute Class(#PRIM\_ATTR.AX\_IN\_INTERFACE) Guid('{BA8C64E0-5654-472F-92DA-255D6DC8B999}') BaseDispId(0) Attribute Class(#PRIM\_ATTR.AX\_EVT\_INTERFACE) Guid('{AEC6E7D2-D3C9-48B0-8056-C96B7BCE13B5}') BaseDispId(0) \*

Define\_Com Class(#prim\_lcol<#skilcode>) Name(#SkillsCollection) Help('co DEFINE\_COM CLASS(#skilcode) NAME(#tmpSkill) reference(\*dynamic) Define\_Com Class(#AXOBJECTB) Name(#AXOBJECTB) Define\_Pty Name(uSkills) Get(\*COLLECTION #SkillsCollection) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(0) Name('uSkills') \* Property Messages

DEFINE\_PTY NAME(uMessages) GET(\*Collection #vMessages) HELP('Me Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(1) Name('uMessag Define\_Com Class(#prim\_lcol<#STD\_TEXTL>) Name(#vMessages) \*Property LastMessage

Define\_Pty Name(uLastMessage) Get(\*auto #vLastMessage) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(2) Name('uLastMe DEFINE\_COM CLASS(#STD\_TEXTL) NAME(#vLastMessage) reference(\*c

\* Property LastStaus

Define\_Pty Name(uLastStatus) Get(\*auto #vLastStatus) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(3) Name('uLastStatus) Define\_Com Class(#STD\_TEXT) Name(#vLastStatus)

\*

\* Personal Details

Define\_Pty Name(uGiveName) Get(\*auto #givename) Set(\*auto #givename) I Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(4) Name('uGiveNa Define\_Pty Name(uSurname) Get(\*auto #surname) Set(\*auto #surname) Help( Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(5) Name('uSurnam \* Address Details

Define\_Pty Name(uAddress1) Get(\*auto #Address1) Set(\*auto #Address1) He Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(6) Name('uAddress Define\_Pty Name(uAddress2) Get(\*auto #Address2) Set(\*auto #Address2) He Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(7) Name('uAddress Define\_Pty Name(uAddress3) Get(\*auto #Address3) Set(\*auto #Address3) He Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(8) Name('uAddress Define\_Pty Name(uPostCode) Get(\*auto #PostCode) Set(\*auto #PostCode) He Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(9) Name('uPostCoce \* contact numbers

Define\_Pty Name(uBusinessPhone) Get(\*auto #PhoneBus) Set(\*auto #PhoneE Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(10) Name('uBusine Define\_Pty Name(uHomePhone) Get(\*auto #PhoneHme) Set(\*auto #PhoneHm Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(11) Name('uHome] \* General

Define\_Pty Name(uDepartment) Get(\*auto #deptment) Set(\*auto #deptment) I Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(12) Name('uDepart Define\_Pty Name(uSection) Get(\*auto #Section) Set(\*auto #Section) Help('Se Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(13) Name('uSection) Define\_Pty Name(uSalary) Get(\*auto #Salary) Set(\*auto #Salary) Help('Yearly Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(14) Name('uSalary) Define\_Pty Name(uNumber) Get(\*auto #empno) Set(\*auto #empno) Help('Nu Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(15) Name('uNumber) \*

Mthroutine Name(uLoadEmployee)

Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(17) Name('uLoadE Define\_Map For(\*input) Class(#empno) Name(#uEmpno)

Change Field(#EMPNO) To(#uEMPNO) Fetch Fields(\*ALL) From\_File(PSLMST) With\_Key(#EMPNO) If\_Status Is(\*OKAY) Invoke Method(#com\_owner.uLoadSkills) Uempno(#empno) Endif

Endroutine

Mthroutine Name(uSaveEmployee) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(18) Name('uSaveE Define\_Map For(\*RESULT) Class(#io\$sts) Name(#O\_STATUS)

UPDATE FIELDS(\*ALL \*excluding #empno) IN\_FILE(PSLMST) WITH\_KE

Set Com(#O\_STATUS) Value(#io\$sts)

Execute Subroutine(SETLASTMSG)

Endroutine

Mthroutine Name(uLoadSkills) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(19) Name('uLoadS Define\_Map For(\*input) Class(#empno) Name(#UEmpno)

Change Field(#EMPNO) To(#uEMPNO) Invoke Method(#SkillsCollection.RemoveAll)

Select Fields(#SKILCODE) From\_File(PSLSKL) With\_Key(#EMPNO) Set\_Ref #tmpskill to(\*create\_As #skilcode) Set Com(#tmpSkill) Value(#skilcode) Invoke Method(#SkillsCollection.Insert) Item(#tmpSkill) Endselect

Endroutine

```
* SETLMSG Saves the last message.
```

\*

\* Get the Message using Built In Function : GET\_MESSAGE \*

\* This must be invoked in a Subroutine, as the messages will be cleared after \* leaving or entering any MthRoutine.

\*

\* This is a temporary error mechanism, a better one is coming soon! Subroutine Name(SETLASTMSG) Define Field(#LASTMSG) Type(\*CHAR) Length(80) Define Field(#RETCODE) Type(\*CHAR) Length(2)

Invoke Method(#vMessages.RemoveAll) Set Com(#vLastStatus) Value(IO\$STS) If\_Status Is\_Not(\*OKAY) Use Builtin(GET\_MESSAGE) To\_Get(#RETCODE #LASTMSG) DoWhile Cond('#RETCODE = OK') set\_Ref #vlastmessage to(\*create\_as #std\_textl) Set Com(#vLastMessage) Value(#LASTMSG) Invoke Method(#vMessages.Insert) Item(#vLastMessage) Use Builtin(GET\_MESSAGE) To\_Get(#RETCODE #LASTMSG) Endwhile Endif

Endroutine

 $End\_Com$ 

↑ Case 3 - Visual LANSA Source

### **Reusable Part AXOBJECTB**

\* Reusable Part AXOBJECTB Function Options(\*DIRECT) Begin Com Role(\*EXTENDS #PRIM\_OBJT) Attribute Class(#PRIM ATTR.AX TYPELIB) Guid('{9606E0AB-DCC2-426D-8ABE-EEFFB05F5F07}') Typelibname('LANSA AXOBJECTB LIB') Attribute Class(#PRIM ATTR.AX CLASS) Guid('{4261A3BF-BD23-4955-997F-FE4E0EA19999}') Progid('LANSA.AXOBJECTB') Attribute Class(#PRIM ATTR.AX IN INTERFACE) Guid('{972B0B94-0740-46D7-AD6F-9BC2142F2A01}') Basedispid(0) Attribute Class(#PRIM ATTR.AX EVT INTERFACE) Guid('{A61429A8-0B19-4788-BF8F-8D075B505EE5}') Basedispid(0) \* employee object Define Com Class(#axobjecta) Name(#Employee) Reference(\*dynamic) \* Collection of employees Define Com Class(#prim lcol<#axobjecta>) Name(#Employees) \* Collection Iterator Define Com Class(#prim lcit<#axobjecta>) Name(#itrEmployees) Reference( Define Com Class(#AXOBJECTA) Name(#AXOBJECTA)

Define\_Pty Name(uEmployees) Get(\*COLLECTION #employees) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(0) Name('uEmploy Define\_Pty Name(uCurrentEmployee) Get(\*reference #employee) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(6) Name('uCurrent

Mthroutine Name(uSetCurrentEmployee) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(4) Name('uSetCurr Define\_Map For(\*input) Class(#std\_num) Name(#index) Define\_Map For(\*RESULT) Class(#prim\_boln) Name(#uDirty)

Set Com(#uDirty) Value(#com\_false) Set\_Ref Com(#employee) To(#employees.item<#index.value>) If\_Ref Com(#employee) Is(\*null) Set Com(#uDirty) Value(#com\_true) Endif

Endroutine

Mthroutine Name(uGetNextEmployee) Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(8) Name('uGetNex Define\_Map For(\*RESULT) Class(#prim\_boln) Name(#uDirty) Define\_Com Class(#prim\_boln) Name(#EmployeeExists)

Set Com(#udirty) Value(#com\_false) If\_Ref Com(#itremployees) Is\_Not(\*null) Invoke Method(#itrEmployees.MoveNext) Result(#employeeexists) If Cond('#employeeexists = #com\_true') Set\_Ref Com(#Employee) To(#itremployees.Current) If\_Ref Com(#Employee) Is(\*null) Set Com(#uDirty) Value(#com\_true) Endif Else Set Com(#udirty) Value(#com\_true) Endif Else Set Com(#udirty) Value(#com\_true) Endif

If Cond('#uDirty = #com\_true') Set\_Ref Com(#Employee) To(\*null) Endif

Endroutine

Mthroutine Name(uGetAllEmployees) Help('Get all employees') Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(1) Name('uGetAllF Define\_Map For(\*output) Class(#prim\_boln) Name(#uDirty)

Invoke Method(#employees.RemoveAll) Select Fields(#EMPNO) From\_File(PSLMST2) Set\_Ref Com(#employee) To(\*create\_as #axobjecta) Invoke Method(#Employee.uLoadEmployee) Uempno(#empno) Invoke Method(#Employees.Insert) Item(#employee) Endselect

Invoke Method(#Employees.CreateIterator) Result(#itrEmployees)

```
If Cond('#employees.itemcount > 0')
Set Com(#uDirty) Value(#com_false)
Else
Set Com(#uDirty) Value(#com_true)
Endif
```

Endroutine

Mthroutine Name(uGetDepartEmployees) Help('Get an individual employee') Attribute Class(#PRIM\_ATTR.AX\_IN\_MEMBER) Dispid(2) Name('uGetDep Define\_Map For(\*input) Class(#deptment) Name(#uDepartment) Define\_Map For(\*output) Class(#prim\_boln) Name(#uDirty)

Change Field(#DEPTMENT) To(#uDepartment) Invoke Method(#employees.RemoveAll) Select Fields(#EMPNO) From\_File(PSLMST1) With\_Key(#DEPTMENT) Set\_Ref Com(#employee) To(\*create\_as #axobjecta) Invoke Method(#Employee.uLoadEmployee) Uempno(#empno) Invoke Method(#Employees.Insert) Item(#employee) Endselect

Invoke Method(#Employees.CreateIterator) Result(#itrEmployees)

If Cond('#employees.itemcount > 0') Set Com(#uDirty) Value(#com\_false) Else Set Com(#uDirty) Value(#com\_true) Endif

Endroutine

End\_Com

↑ Case 3 - Visual LANSA Source