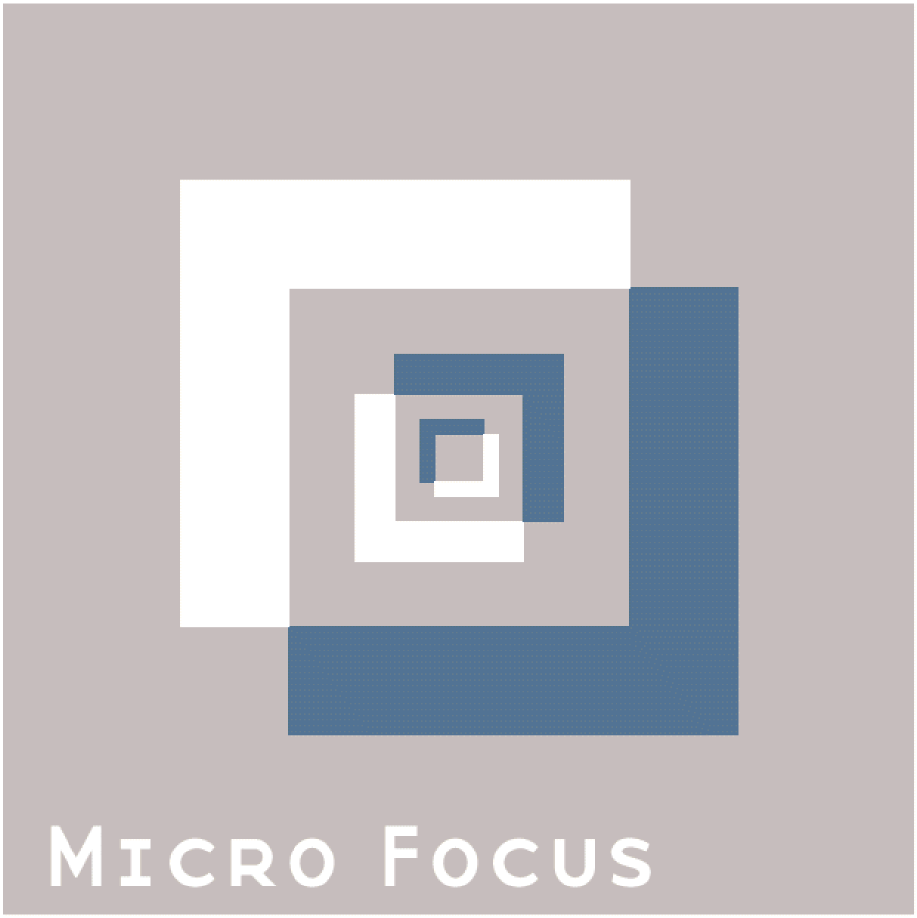
Issue 8e, March 2002



Net Express®

Dialog System User's Guide

MICRO FOCUS

Micro Focus International Limited has made every effort to ensure that this book is correct and accurate, but reserves the right to make changes without notice at its sole discretion at any time. The software described in this document is supplied under a license and may be used or copied only in accordance with the terms of such license, and in particular any warranty of fitness of Micro Focus software products for any particular purpose is expressly excluded and in no event will Micro Focus be liable for any consequential loss.

20020513085721

Table of Contents

Part 2: Advanced Features

Part 3: Programming Tutorials

Preface

This user guide introduces you to the subject of graphical user interfaces and how to use Dialog System to full advantage. It gives you a tour of the major menu choices, taking you through creating and using a simple screenset, showing you the important features and demonstrating the general sequence of events in developing an application. The user guide also covers the more advanced features. There are plenty of examples and sample programs, as well as hints and tips on getting the most out of Dialog System.

Audience

This book is for all programmers and system designers using Dialog System, whether experienced with Micro Focus's earlier Dialog System products or completely new to Micro Focus. It assumes you are familiar with the general concepts of business computing and of using Microsoft Windows.

Related Publications

- Help containing ***Dialog System Reference***
- On-line help for Net Express and for other components of your COBOL system

Notations and Conventions

- **Enter** refers to the carriage return or Enter key. Where commands to be typed are shown, the Enter key is not explicitly shown; it is treated as implicit that Enter must be pressed at the end of the line.
- Hexadecimal numbers are enclosed in quotation marks and preceded by a lower-case "x" or "h"; for example, x"9D", h"03FF". The "x" is used when the hexadecimal number represents a character string; the "h" when it represents a numerical value.
- PIC X is used rather than PIC 99 with the COMP-X and COMP-5 data types. Unlike PIC 99, PIC X shows the length of the data item and so demonstrates more clearly the use of COMP-X, which is to define a binary item of the specified number of bytes.
- Keytops and menu choices are emboldened within the text.
- Side headings are used to indicate environment specific information. For example:

Windows: Windows specific information.

- In some environments, you might notice that what appears on your screen differs in minor ways (for example, version numbers) from that illustrated in this book. This will not affect the operation of your software.
- The keys described in this book are not available in all environments. When there is a reference to use of a key such as a status or function key, this refers to the logical press and release of this key, rather than physical keystroke. If your environment does not support the key given, please refer to your accompanying *Release Notes* for the equivalent key.
- The term "window" refers to a delineated area on the screen, normally smaller than the full screen. The term "Windows" refers to Microsoft Windows 95 or later.
- On-line help is not described in the documentation. Select **Help** from the menu or press the **Help** button on a dialog box to see context sensitive help information.

The notation used to describe the format of command lines is as follows:

- Words printed in italics are generic terms representing names to be devised by you.
- Material enclosed in square brackets [ ] is optional.
- When material is enclosed in braces { }, you must choose from the options within them. If there is only one option in the braces, the braces indicate repetition.
- The ellipsis (. . .) follows { } or [ ] and means you can repeat the material in the { } or [ ]. The number of repetitions allowed is unlimited unless otherwise stated. If the ellipsis is used with [ ] the material can be omitted altogether.
- If a command line does not fit across the page, it is continued on the next line; the continuation line is indented.
- Command line options can be specified as /option or -option.

Part 1: Introduction

This part contains the following chapters:

- Chapter 1, “The Graphical User Interface”
- Chapter 2, “Introduction to Dialog System”
- Chapter 3, “Creating a Data Definition and Screenset”
- Chapter 4, “Window Objects”
- Chapter 5, “Control Objects”
- Chapter 6, “Using Dialog”
- Chapter 7, “Using the Screenset”
- Chapter 8, “Windows GUI Application Wizard”

1 The Graphical User Interface

This chapter introduces the graphical user interface (GUI) and explains how to use it.

Dialog System provides software that enables you to build a usable GUI quickly and easily. It has an integral testing facility, which enables you to test and prototype the interface before you start to write the program that will use the interface. Some important features of Dialog System are:

- You can create a user interface that is independent of your application program and main program logic.
- You can define and change the user interface without affecting the application program.
- You can create several different user interfaces for the same program.
- You can prototype and test the interface without a supporting COBOL program.

  This means that you can enter data, receive output, and navigate between windows solely within Dialog System, without involving the program.
- All interaction with the user at run time is handled by calls to Dialog System from your application program.

  This means that your application program is smaller and easier to maintain.
- Dialog System helps to create the program by providing a ready-coded data structure for communication between the program and the user interface.
- Dialog System is portable among different environments.

Terminology

The *Glossary* in the Help fully explains the terms used in the Dialog System documents. Where two terms exist for the same item, the glossary explains both and notes the term preferred in Dialog System.

1.1 Why Use a GUI for Your Application?

The aim of a GUI is to provide an interface that is:

- Easy to learn.
- Consistent across applications (needs learning only once).
- Easy to use.

The GUI encourages users to concentrate on their tasks, rather than on the functions provided by the application. A GUI is intuitive, and encourages users to experiment and explore. It also forgives mistakes. This means that users can learn in their own way and at their own pace.

When they revisit the GUI, they are more likely to remember how to use it. If they don't remember, it doesn't matter because they can experiment all over again.

Character-based application interfaces use a hierarchy of menus. These require users to navigate through the different levels to the required function. Alternatively, users can memorize obscure shortcut keys.

A GUI application approaches the task to be done more directly. Users select an object and then an action. This reduces the number of menu levels. By using the point and click actions of the mouse, users can move easily through the interface until they can do the required task.

A GUI provides contextual help, with links to related areas. This makes it easier for users to acquire the background knowledge they need to do a task, without using more complicated help facilities that are supplied with character based systems, such as navigating through a hierarchical help tree.

1.2 How Dialog System Helps

Dialog System helps you with five phases in the life cycle of the user interface:

- Defining the user interface.

  Use Dialog System to define the data you want your application to use and its presentation on screen. This includes how the user will input data and navigate around the interface. This information is called the screenset and is saved as a file with a **.gs** extension.

- Prototyping and testing.

  Use Dialog System's testing facility to enter data and navigate around the interface as if the program were running. Simulate outputs and monitor what is returned to users.

- Integrating with the calling program.

  When the interface has been thoroughly tested, use Dialog System to generate a copyfile containing details of the data structure and the run-time interface.

- Running the application.

  Use Dialog System in run-time mode to handle dialog with the user and use the screenset until control returns to the calling program.

- Maintaining the application.

  Use Dialog System to change and customize the user interface without affecting the calling program. Conversely, you can change the calling program, without needing to change the user interface. Change both only if you need to change the data items passed between the user interface and the program.

1.3 Using a GUI System

Dialog System is designed for use with a mouse. You use the mouse to select, move and size the objects on the screen. You can also use the keyboard to do this, but you are likely to find the mouse more convenient. The keyboard is used to enter text and numeric data.

The following sections explain how to use the mouse, windows and control objects for those who are unfamiliar with a GUI environment.

1.3.1 Mouse Actions

The mouse operates a pointer that shows your position on screen. Also on the screen there is a selection cursor, which shows your position within a window.

The mouse pointer can move across window borders, and will change shape. For example, in a text field, it appears as an I-beam pointer, showing you the exact place where any text will be added. At the corner of a sizable window, it appears as a double headed arrow, and while you wait for an action to complete, it appears as a clock or an hourglass (depending on your current environment). The shape of the mouse pointer should give you some clues as to what you can do at a particular place on the desktop.

Before you perform any of the mouse actions, move the mouse pointer to the appropriate position on the screen. For example, if you want to select a menu option, move the pointer until it is over that option.

You can use the mouse to operate the various buttons on the screen, as well as to select objects or items and scroll through information on the screen.

In the following list of mouse actions, the mouse button to use for each action is not specified because some behavior is specific to an underlying operating system:

- Click - Press and release the mouse button.

  Selects an item or object, presses a push button or radio button, toggles a check box or a checkmarked choice.

- Click and drag (also called rubber band) - Press and hold down the mouse button, move the mouse pointer in the desired direction or over the desired selection, then release the mouse button.

Draws a box around an area to select a number of items, moves a selected object, or changes an object boundary in a sizing operation.

- Double-click - Press and release the mouse button twice.

  On a Dialog System object, displays that object's Properties dialog box.

- Release - Let go of the mouse button which has been held down.

  Stops scrolling, stops dragging the object in a move operation, stops moving the window or object boundaries in a sizing operation.

Other mouse button behavior relating to Dialog System, for example Move or Select, is configurable. You can attach specific behavior for use in Dialog System which provides various standard patterns for these. What you choose to use depends on your operating environment, the software you are familiar with, and the type of mouse you have (two button or three button).

1.3.2 Windows and Menus

The desktop is the working area on your screen. A window is an object that represents all or part of the screen. You can create windows on the desktop that extend beyond the edges of the screen. You can also move windows around on the desktop.

The first window you create is called the primary window. It is a child of the desktop, because there are no other windows above it. This means that the desktop is its parent. You may create more than one primary window.

Any primary window can have child windows, called secondary windows. These in turn can have other secondary windows.

Primary and secondary windows are fully explained in the chapter *Window Objects*

The main menu bar appears on your desktop when you start up Dialog System.

You can pull down submenus from the menu bar by clicking on the choice required. Clicking on another choice while a submenu is pulled down simply closes the current menu and pulls down another.

Sometimes, choices are not available until you have performed a particular action (for example, you cannot create a control object such as a push button, until you have created a window or dialog box to put it into). If a choice is not available, it is displayed in a pale color instead of the deeper color of the available choices. The choice is said to be disabled. If you click on a disabled choice, you may hear a beep, but nothing else happens.

Menu choices can:

- Pull down additional menus. These are identified by arrows following the choice.
- Cause a secondary window or a dialog box to be displayed. These are identified by an ellipsis (...) following the choice.
- Be checkmarked choices. These have two states, on or off. When you click on the choice, you reverse its state. If it is on, a checkmark appears in front of it. These choices are sometimes called toggles.
- Cause an action to be performed directly. These have no special identification.

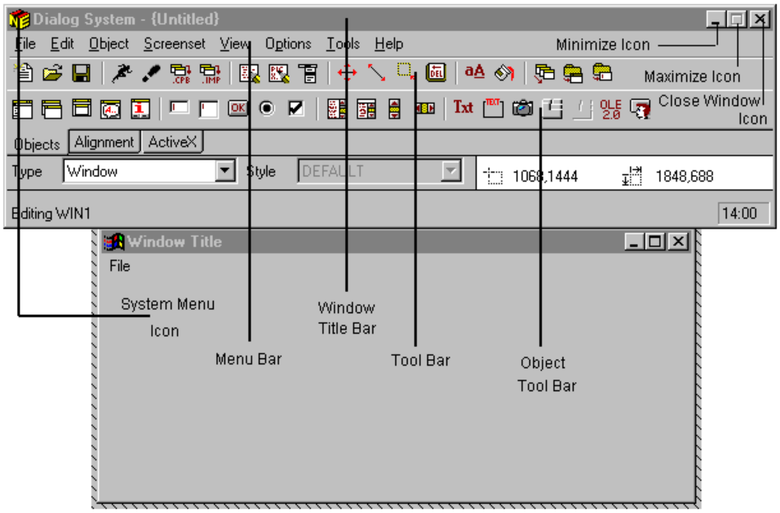
Any one menu choice can only have one of the actions described in the preceding list, and is decorated (for example with an ellipsis or a checkmark) appropriately.

1.3.2.1 Manipulating Windows

This section describes various ways of manipulating windows using the mouse and the icons that form part of most windows.

Figure 1-1 shows a window with labels to identify its components.

Figure 1-1. Window Components



When you are not using a primary window, you can reduce it to a small symbol, called an icon, on your taskbar.

If the window has a minimize icon, you can minimize the window using the mouse. Click on the minimize icon, which is the leftmost icon in the top right corner of the window. Double-click on the window icon to restore the window (see the section *Mouse Actions* earlier in this chapter).

You can also move a window and change the size of the window. These operations are best done with the mouse. To move a window, put the mouse pointer in the window title bar, click and drag the window to a new location, then release. The button you click to do this depends on how you have your mouse set up.

Similarly, to size a window, put the mouse pointer in the border at the side or corner of the window and click and drag it. The mouse pointer changes shape when it is in the right place to size the window. As you drag the window side or corner, the window changes shape. When the window reaches the required size and shape, release the mouse button.

To close a window, double-click on the system menu icon at the top left corner of the window.

To move between windows on the desktop, click on part of the window you want to move to. (If you click on any of the icons in the title bar, you activate the icon.)

If the window is not visible, select the required window from the taskbar and switch to it.

The current window (that is, the one that is active) has a colored title bar.

For more information on manipulating windows, refer to your operating system's documentation.

1.3.3 Dialog Boxes

A dialog box is a type of window that enables users to enter data of some kind. It cannot be resized and cannot have a menu bar. You can add any of the Dialog System control objects (such as a push button or a list box) to it. A dialog box can be created by another window, or by the desktop. (It is called a child of the window that creates it, and that window is called the parent of the dialog box.)

Dialog boxes stay in place until they are closed by user actions. Typically, a user clicks a push button to cause Dialog System to accept user actions (for example, the **OK** button to accept a selection from a list). Alternatively, a user may click a push button to cause Dialog System to ignore user actions (for example, the **Cancel** button to close a dialog box, ignoring any input).

Dialog boxes are often used to create File Selection boxes. These enable users to choose from a selection of files or other items and to enter their selections in another field.

1.3.4 Message Boxes

A message box is another type of window that appears on the screen to give users a message. It can contain only text and graphics to give a message, and push buttons to respond to the message. It must be explicitly cleared by the user clicking one of the push buttons.

You can use message boxes to give warnings when incorrect data has been entered, or to ask users to confirm that they want to continue with a potentially destructive action (such as deleting an object).

1.3.5 Controls

Controls are Dialog System objects that you can add to windows or dialog boxes to enable users to interact with the application. Control objects can only be placed inside a window or dialog box. Dialog System provides a wide range of control objects such as entry fields, push buttons, list boxes, selection boxes and bitmaps. See the chapter *Control Objects* and the Help for full information on the control objects available.

1.3.6 Selecting Objects

In a GUI system, the application is driven by users selecting objects and then selecting actions to be applied to the objects.

Dialog System provides the following selection methods that are applicable to most objects:

- To select a single object, move the mouse pointer to the object and click.
- To select several objects with the mouse, one of the mouse buttons must be set to Select (see the section *Mouse Actions*).

  If you hold down the shift key, you can select multiple individual objects by clicking on them, rather than by clicking and dragging.
- To select objects using the menu or keyboard, use **Select area** or **Select all** on the **Edit** menu. **Select area** selects one or more objects. **Select all** selects all objects in the current window or dialog box.

Dialog System provides the following selection methods applicable to specific objects:

- To select a menu choice, either move the mouse pointer to the choice and click, or select the choice using the cursor arrows and press **Enter**.
- To select an entry in a selection box, scroll through the list in the box until the desired entry appears, move the mouse pointer to it and click. If you double-click on a list item, the default action is performed. Clicking on another item cancels the first selection and selects the new one.
- To select an entry in a list box, scroll through the list in the box until the desired entry appears, move the mouse pointer to it and click. If you double-click on a list item, the default action is performed. A list box can be set up to enable you to perform a single selection only, to select multiple items, or to select multiple adjacent items.
- To select a radio button choice, click on the button. It is filled in to show it has been selected. If another radio button in the same control group was selected, it will be deselected. (Only one radio button can be selected in a control group.)
- To select a check box choice, click on the check box. The state of the check box is reversed - if it was selected, it is now deselected, and if it was not selected, it is now selected. You can click on any number of check box choices.
- To select a push button action, click on the button. The button changes appearance to look as if it is pushed in. Clicking the button starts the action.

1.3.7 Scrolling

Some windows contain scrollable areas. These are indicated by scroll bars at the right hand edge (for vertical scrolling) and the bottom edge (for horizontal scrolling). Each scroll bar contains a slider which moves along the bar, and arrows indicating the direction of scrolling.

- To scroll line-by-line, click on the scroll arrow that points in the direction you want to scroll.
- To scroll a screen at a time, click on the scroll bar itself, next to the slider. Scrolling is towards the point where you clicked.

- To scroll more than a screen at a time, click and drag the slider in the appropriate scroll bar in the desired direction. Release the mouse button when you reach the desired part of the window.

1.4 Further Information

The next chapter *Introduction to Dialog System* explains the basic concepts that you need in order to start using Dialog System.

2 Introduction to Dialog System

The previous chapter described the advantages of using a graphical user interface (GUI). This chapter introduces you to the basic concepts which you need in order to use Dialog System when creating GUIs for your COBOL applications. These include:

- The benefits of using Dialog System.
- An overview of the constituent parts of Dialog System.
- The steps that you need to follow to create an application with Dialog System.

2.1 Benefits of Using Dialog System

Dialog System offers you the following benefits:

- Independence of the user interface from the main program logic.

  When the program requires interaction with the user at run time, it calls Dialog System to handle this. This independence encourages well-structured programming.
- Independence of the user interface from the program.

  You can create several different user interfaces for the same program.
- Ease of defining graphical objects.

  Dialog System provides a library of graphical objects that you can tailor for your application and manages these objects' behavior, including stacking and unstacking windows. Dialog System also deals with screen handling, both during definition and at run time.

- Complete handling of input, output and navigation between windows.

  You can instruct Dialog System to handle user input and output and decide what to display to the user, using a simple set of instructions called dialog.

- Automatic Data Block generation.

  Dialog System can supply your calling program with the definitions for the Data Block that are needed to interface with Dialog System at run time.

- Validation capability.

  Dialog System handles most validation of input and can deliver error messages to users. The calling program can also use these error messages if required.

- Prototyping and testing.

  Dialog System can run the interface in a similar way to the calling program, which gives a high degree of confidence in the validity of the results. You do not need to wait until you have a complete interface as Dialog System enables you to test small parts of the interface as you develop them.

2.2 Overview of Dialog System's Capabilities

Dialog System enables you to design and edit windows and dialog boxes for display on a Windows system and to pass data between this interface and your application program. The two aspects of Dialog System which enable these functions are:

- The definition software, which enables the creation and tailoring of the items to appear on your user interface.

- The program **Dsgrun**, which communicates with both your interface and your application program and controls the run-time behavior of your interface components.

This section describes in outline what the definition software is and how it enables you to create a user interface, covering:

- The design of your interface and its elements.
- How to activate these elements using dialog.
- How you link the above parts to your calling program using the Data Block.

2.2.1 Designing Your Interface

The first step when you develop a Dialog System application with a GUI is to design a data model. The model defines the data which is to be both captured by the user interface and provided by the application to be presented to the user.

This data model provides the basic data items that must be passed to the calling program. You must also decide on any validation criteria for these items.

Some of the desirable features you should consider for your user interface are:

User controlled
Users are in control of the interface. They determine the course of action of the application.

Easy to use
The user can move easily and naturally around a window and between windows. The information displayed is well organized, easy to read, and appears uncluttered.

Consistent
The user interface is consistent internally as well as across applications. For example, the **Exit** choice on a window is always the last option in the **File** menu. Another example is that any delete action requiresadditional confirmation.

Helpful
Users are given clear, informative feedback on where they are in the system, what actions they have taken, and what action they should take next. It is better for the user to select an item from a list rather than enter the item in a data entry field.

Forgiving	User actions are easily reversed. Error messages explain what the error was, why the error occurred, and possible corrections to the error.

Efficient	System response times are kept as short as possible.

Complete	If a user has a list of choices, all choices are valid and available to the user.

Logical	Choices on a menu are logically organized, with the most frequently chosen item listed first or with the items listed alphabetically.

You also need to consider how the user interacts with the data - the user interface.

The user interface you build can take many forms, ranging from simple menu selection and data entry fields to fully developed windows complete with menu bars, radio buttons and other controls.

The level of sophistication of the interface you choose depends on many factors including:

- Who the users are and their level of training.
- The hardware and software resources available.
- The complexity of the task to be performed.

In all cases:

- The user determines the normal course of action.
- The user, rather than your program, is in control of the interface.
- The user interaction with the computer should be as simple and natural as possible.

The basic visual elements which you define for your user interface are called objects. These objects, such as windows, dialog boxes and push buttons, appear on an area bounded by the screen, in which everything else appears.

In Dialog System, there are two main categories of objects:

- Window Objects
- Control Objects

2.2.1.1 Window Objects

These are the most basic (and often the most important) objects. They can be primary or secondary and are very similar to dialog boxes and message boxes. The appearance of windows and dialog boxes is the same at both definition and run time.

You can define window objects at any time and place them anywhere on the desktop. You can move these objects by dragging the title bar and you can size windows using their sizable borders. When selected, you see them surrounded by a colored, shaded border.

For further information, see the chapter *Window Objects*.

2.2.1.2 Control Objects

All other objects are control objects. These are the objects that appear in the windows and include entry fields, push buttons, radio buttons, check boxes and list boxes.

You can define other control objects which extend Dialog System's default range of objects. When you define such a control object, Dialog System can generate a tailored controlling program for the control.

The two types of additional controls that you can define are:

- User controls.

  These enable the definition of a container or outline for objects not paintable by Dialog System.

  At definition time, the generated controlling program name appears inside the control's outline. Specifying a meaningful name helps you to identify the control and its purpose in the GUI.

- ActiveX controls.

  These are third-party supplied controls. You need to create the program code to create and manipulate them at run time. When selected, you see them as they appear at run time.

Both of these types of controls are described in the chapters *Control Objects* and *Programming Your Own Controls*.

2.2.1.3 Object Properties

Every window object and control object has properties, for example, background color. You can change these properties to specify the object's appearance and behavior.

You can use the default properties of an object by selecting **Include** on the **Options** menu and selecting **Auto properties**. In this case Dialog System uses the default property values for that object and does not display the Properties dialog box. You can change the properties of the object using the Properties dialog box by selecting **Properties** on the **Edit** menu or double-clicking on the object.

If **Auto properties** is not selected, Dialog System immediately displays the Properties dialog box for an object, so you can set the properties. If you click **Cancel** in this dialog box at this stage, the object is removed. If you click **OK**, the object is defined using the properties displayed in the dialog box.

If you define a message box, bitmap, ActiveX or User control, the Properties dialog box is always displayed, regardless of the setting of **Auto properties**.

2.2.1.4 Working with Objects

To work with an object, you must make it the current object. When you define an object initially, Dialog System automatically makes it the current object.

2.2.1.5 Naming Objects

A common property of all objects is a name. For some objects, it is required, and for some it is optional. If you give an object a name, that name must be unique.

A good programming practice is to set up a naming convention for the variables in your COBOL program. A consistent naming convention enables you or another developer to understand your application more easily.

We also recommend that you establish a similar naming convention for the objects in your user interface.

One fairly simple naming convention consists of the concatenation of:

windowname-objectinfo-objecttype

where the concatenated parts are:

windowname The name of the window the object is located on, or if the object is itself a window, just the window name.

objectinfo Some identifying information about the object. The information should uniquely identify the object on this window, for example, the text that appears on a push button.

objecttype An abbreviated name that identifies the type of object. For example, you could use *win* for window, *pb* for push button, *db* for dialog box and so on.

Using this convention, a window used to gather information about a new employee could be named NEW-EMPLOYEE-WIN. A push button on the same window could be named NEW-EMPLOYEE-OK-PB. An entry field on the window that is used to input salary information could be named NEW-EMPLOYEE-SALARY-EF.

2.2.2 Using Dialog

A user interface is more than just a graphical display. A complete specification also describes how the user and computer interact and how the user interface software interacts with the application software.

Once you have defined the appearance of the display, you must define this run-time interaction between the user and the machine. This interaction is called dialog. Dialog consists of events and functions. When an event occurs, the functions associated with the event are executed. Events can be caused by a key on the keyboard being pressed, or a menu choice or an object being selected.

For example, a Dialog System event such as BUTTON-SELECTED occurs when a user selects a push button (with the mouse or keyboard). If the button selected is **Enter**, the function associated with it might be CREATE-WINDOW, to create a new window for the user to enter more information.

Dialog System lets you create or customize the dialog between the user and the display objects.

For more information on dialog statements, see the chapter *Using Dialog* and the topic *Dialog System Overview* in the Help.

2.2.2.1 Events

An event signifies some change in the user interface: events happen to objects such as windows, bitmaps, list boxes and buttons. There are many different operations which can trigger an event, such as:

- The user pressing a key.
- A click on a mouse button when the mouse pointer is located over a push button.
- A window or control receiving focus.
- A validation error occurring.
- The user moving the slider on a scroll bar.

Intrinsically, all events are the same. However, for convenience, they can be divided into three types:

- Menu Events - An option is selected from a menu, for example, when you choose **Exit** on the **File** menu.
- Object Events - Objects are activated, such as windows, push buttons, radio buttons, list boxes and dialog boxes.
- Keyboard Events - Events occur when a key is pressed on the keyboard.

When an event occurs, Dialog System searches first in the relevant control dialog, then the relevant window dialog, then the global dialog. Defining an event is part of the process of creating a Dialog Definition. Events are defined by associating Functions with them.

2.2.2.2 Functions

Functions are instructions to Dialog System to do something. They are associated with events and operate when:

- An event they are listed under occurs.
- A procedure they are listed under is executed.

Dialog System has a comprehensive set of functions. Some are specific to moving around screensets, for example SET-FOCUS or INVOKE-MESSAGE-BOX. Some are similar to other programming language instructions, for example MOVE data from one data item to another.

You can find a complete description of the functions in the topic *Dialog Statements: Functions* in the Help.

2.2.2.3 Procedures

A procedure is an arbitrary name with a set of functions listed under it; in other words, a subroutine. You can think of a procedure as being like an event.

2.2.3 Using the Data Block and Screenset

Dialog System creates a file, called a screenset, which holds the definitions of all the windows and dialog boxes as you create them for your interface. It is called ***filename*.gs**. This includes any on-screen objects which you may need in your interface, for example, entry fields.

When you run your interface, objects such as entry fields contain data which you need to pass between the interface and your application program. You do this by associating your on-screen objects with data items defined in your application program. These data items are called master fields in Dialog System. You make the association by selecting the appropriate master field when editing the control's properties.

These data definitions are held in the screenset file and form the Data Block. When the generated program calls Dialog System's run-time support module, **dsgrun**, it passes the Data Block as a parameter.

The screenset file holds:

- The data block.

  The data definitions for all of the input and output.
- Objects for windows and dialog boxes (including validation rules).
- The actions associated with those objects.

  These are called dialog events and functions, and you define them.

2.3 Steps for Creating an Application Using Dialog System

To start Dialog System:

- Select **Dialog System** on the IDE **Tools** menu.
  - or -
  Select **Net Express Command Prompt** and enter **dswin**.

The steps that you need to follow to create an application with Dialog System are:

1. Define data and validations.

   See the section *Steps for Creating a Data Definition* in the chapter *Creating a Data Definition and Screenset* and the topic *Data Definition and Validation* in the Help.
2. Define windows, dialog boxes and message boxes.

   See the chapters *Window Objects* and *Control Objects*.
3. Define control objects (such as entry fields, text fields and radio buttons) and place them on the windows and dialog boxes.

   See the chapters *Window Objects*, *Control Objects* and *Using Dialog*.
4. Save your screenset.

   It is a good idea to save a screenset whenever you have finished a stage in the definition process. The first time you save a screenset, use **Save As**. Dialog System displays a dialog box for you to enter the filename and directory you want to save under.

After saving the sample screenset once, you can use **Save** whenever you want to save the screenset. If you want to try out different versions of the screenset, use **Save As** with a new name to create another version.

5 Test your screenset.

Examine the behavior of the screenset using the Screenset Animator. See the section *Screenset Animator* in the Help.

6 Define dialog.

Define object dialog that controls the behavior of objects.

7 Test the screenset again.

See the section *Screenset Animator* in the Help.

8 Change the screenset as required.

Go back to earlier stages of this process.

9 Generate the COBOL copyfile from the screenset.

See the section *Steps for Creating a Data Definition* in the chapter *Creating a Data Definition and Screenset*.

10 Write the COBOL application program with the necessary calls to the Dialog System run-time system.

See the chapter *Using the Screenset*.

11 Animate and test the COBOL program.

See the topic *Debugging* in the Help.

12 Package your application.

See the topic *Compiling and Linking* in the Help.

You do not need to follow this order strictly. You can define objects before you define data if you wish, but you cannot define object dialog until you have defined an object to which you can attach the dialog.

3 Creating a Data Definition and Screenset

The following sections describe the steps needed to create an application with Dialog System.

3.1 Designing a Data Model

As you have seen in the chapter *Introduction to Dialog System*, the first step in developing a Dialog System application is to design a data model which you can use to create the data definition. This data model defines the basic data items your screenset must pass to the calling program when you run your interface.

You should base the data definition on the data model of your calling program. Before you start to create the data definition, identify the data to be captured by the user interface. This data becomes the basic data items your screenset must pass to the calling program. Also, decide on any validation criteria for these items.

3.1.1 Defining Data and Validations

In this step of the development cycle, you first create the data model for both input and output data. This should include data characteristics such as:

- Data type, for example integer, string, or computational.
- Data size.
- Validation requirements .
- Relationships among the data items.

When you have created the model, you can begin to define the data that passes between Dialog System and your program.

There are also some design considerations concerning the whole of your screenset. See the section *Controlling the Use of Screensets* in the chapter *Using the Screenset* for further details.

3.2 The Data Definition

To pass data between the interface and your application program, you associate some objects, such as entry fields, with data items (master fields) defined in your application program. These are the data definitions.

A master field holds the data that users enter in an entry field. You make the association by selecting the appropriate master field when editing the control's properties.

3.2.1 Prompted and Unprompted Mode

Dialog System allows for inexperienced users by providing a prompted mode of data entry. The differences are:

- Prompted mode prompts you for the type of line to enter, either a comment, field, group start or end.
- If you make entries using unprompted mode, they are formatted when you finish entering the line. You must separate the items with spaces.

3.2.2 Comments

It is good practice to place comments in your data definition. Any line starting with an asterisk character is treated as a comment, with no validation or reformatting performed. For example:

```
* This is a comment line
```

3.2.3 Steps for Creating a Data Definition

Follow the steps below to create a data definition:

- Identify the data items to include in your data definition.
- Assign a data type code.
- Use data groups.
- Look at data dependencies.
- Specify validation criteria and associating error messages.

3.2.4 The Data Block

The Data Block contains the data items that you define in the Data Definition window of Dialog System. Each screenset has a unique Data Block that is incorporated into the calling program as a copyfile.

At run time, the calling program moves user data to this record before it calls Dialog System. User input placed in the Data Block by Dialog System can be used by the program when control is returned.

Related data items can also be placed in groups, and mapped onto objects such as list boxes.

You can view, define, and edit the Data Block in the Data Definition window. To invoke the Data Definition window:

- Select **Data block** on the **Screenset** menu in the main window.

A simple example of a Data Block is shown below:

Fieldname	**Format**	**Length**	**Validations**
`GROUP-ITEMS`	Group	10	
`FIELD-1`	9	4.00	R
`FIELD-2`	X	10.00	R
`FIELD-3`	9	6.00	R
`GROUP-POSITION`	9	2.00	

Fieldname	Format	Length	Validations
S-FIELD-1	9	4.00	
S-FIELD-2	X	10.00	
S-FIELD-3	9	6.00	
COMMENT	X	40.00	
COMMENTS-TITLE	X	25.00	
ARRAY-SIZE	9	2.00	
OBJECT	OBJ-REF		

The format of the Data Block is controlled by Dialog System.

- Entries made using prompted mode are formatted automatically.
- Entries made using unprompted mode are formatted when you finish entering the line.
- Each item in a line must be separated by a space.

If an error occurs, you will be prompted to re-key the line. All entries in the data definition are converted to upper case.

Note: OBJ-REF does not require an entry under length, as it is fixed length.

The data definition is copied when you generate the COBOL copyfile for your screenset. This copy file, named ***screenset-name.cpb*** forms the Data Block which is passed between the calling program and Dialog System. When you run your screenset, the calling program moves data to the Data Block before it calls Dialog System. Dialog System can move data to the Data Block before it returns control to the calling program.

3.2.4.1 Data Block Copyfile

The Data Block copyfile contains a description of the screenset data items and associated field numbers.

The copyfile is a COBOL record definition that is incorporated into the calling program when it is compiled and is entirely dependent on the data items you defined as part of the screenset. If these change, your program will need to change to match.

A checking mechanism is used to ensure this. The format of the Data Block copyfile is as follows:

- The Data Block corresponds to a level-01 record in a COBOL program Working-Storage Section.
- Single data items correspond to level-03 data items in the record.
- Data groups correspond to level-03 group data items.
- Data group items correspond to level-05 data items.

3.2.4.2 Data Items

All data items defined in the data definition are global in the screenset and can be freely referenced in the screenset dialog.

Typical data items might map onto an entry field or a list item. Data items are categorized according to type, and can be placed in groups and mapped onto an object such as a list box. Data items can also be set up as flags, which can be referenced by dialog or can carry values to your calling program.

Each data item in the Data Block must have the following:

- A name of up to 30 bytes in length.
- A data type code to enable the calling program to determine the data type. The data type code can be any of the following:

X	Alphanumeric.
9	Numeric (up to a maximum length of 18 characters), including integer and decimal places. The number of decimal places after the point must not exceed 9.
A	Alphabetic.
S	Signed numeric.
C	Computational (number of bytes must be 1.0, 2.0, 4.0 or 8.0). Computational data items cannot be displayed or tied to any entry fields.
C5	COMP-5 (number of bytes must be 1.0, 2.0, 4.0 or 8.0). COMP-5 data items cannot be displayed or tied to any entry fields.
N	DBCS (N). Double-byte Character Set.

G	DBCS (G). Double-byte Character Set.
OBJ-REF	An object reference, used to store a reference to a class or instance object.

- The size of the data item. The meaning of this item depends on the data type selected, as follows:

Data type code	**Meaning of size field**
X or A	Number of characters.
9 or S	Number of digits before and after the decimal point.
C or C5	Number of bytes stored. The size determines the range of numbers that can be stored: 1.00 to 255 2.00 to 65, 535 4.00 to 4,294,967,295 8.00 to 18,446,744,073,709,551,615 Note that in some circumstances, Dialog System handles only the last 18 digits of a value.
N or G	Number of characters. Each character occupies two bytes.
OBJ-REF	The size of object references is fixed. You cannot specify a size for an object reference.

The following examples show the maximum and minimum ranges you can enter for numeric and signed numeric data items:

```
LARGEST-VALUE            9     18.0
SMALLEST VALUE           S     0.9
```

The following example shows a Data Block entry for a numeric data item with three places before the decimal point and two after:

```
ARBITRARY-DATA-NAME      9    3.2
```

The appearance of a data item is controlled by the picture string of the presenting entry field.

The following example shows how to define an object reference:

```
MAIN-WINDOW-SBAR-OBJREF       OBJ-REF
```

3.2.5 Using Data Groups

Data groups are used for multiple occurrences or repetitions of a set of homogenous data items. The data items in a data group are referenced by a subscript in the Data Block and the calling program. Data groups are seen by the calling program as contiguous data items, each item being accessed relative to its position in the group using the subscript as an index. If a data group is defined as having only one repetition, the data items will not have subscripts in the calling program (but you must still use subscripts in Dialog System).

A group entry consists of a group name and a number of repeats, for example:

```
GROUP-1                     150
```

Data items belonging to a group are listed under the group name and indented, for example:

```
GROUP-1                     150
     DATA-ITEM-1             X    10.0
     DATA-ITEM-2             X    10.0
     DATA-ITEM-3             X    10.0
```

It is not possible to change the order of data items in the Data Block other than by deleting and redefining them or using the **Copy**, **Paste** and **Delete** functions from the **Edit** menu in the Data Definition window.

3.2.6 Dependencies

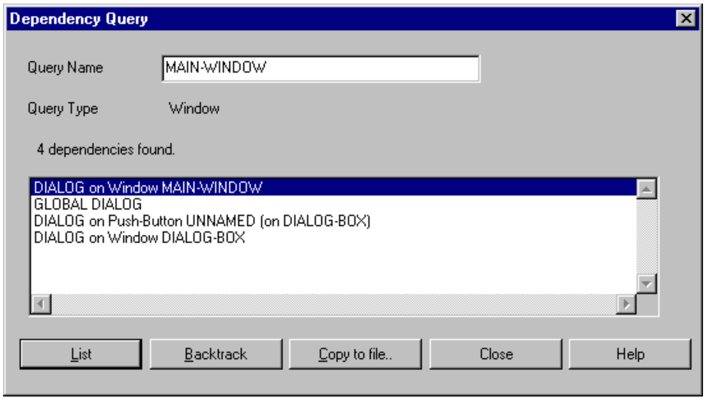
A dependency is the term used to describe the relationship between data items, objects and dialog. Dependencies fall into two categories:

- Where the data item is dependent on an object because it is the Master Field of the object.
- Where the data item is dependent because it is referenced by dialog.

You can query the dependencies of any named object by selecting **Dependencies** from the **View** menu in the main window or by selecting **View Dependencies** from the **Options** menu in the Data Definition window.

The Dependency Query dialog box showing the dependencies for the Customer sample screenset is shown in Figure 3-1.

Figure 3-1. ***Dependency Queries Dialog Box***



Query Name MAIN-WINDOW

Query Type Window

4 dependencies found.

DIALOG on Window MAIN-WINDOW
GLOBAL DIALOG
DIALOG on Push-Button UNNAMED (on DIALOG-BOX)
DIALOG on Window DIALOG-BOX

If you query dependencies from the Data Definition window, the dialog box describes dependencies for the data item on the line currently highlighted in the Data Definition window. If you query dependencies from the main window, the dialog box describes the dependencies for the object highlighted in the main window.

When you are in the Dependency Query dialog box, you can list the dependencies of a data item or object displayed in the dialog box. Highlight the relevant line, then click **List**. This feature enables you to follow a chain of dependencies if you need to.

3.2.7 Validating User Data

Dialog System enables you to validate user data input via entry fields against specified criteria.

Validation only occurs when you use the VALIDATE function in your dialog. For example, to validate all entry fields in the main window, you can use:

```
VALIDATE MAIN-WINDOW
```

If you added any selection boxes to the main window, this function would also validate these. You can also validate a single entry field or selection box.

If a validation fails, a VAL-ERROR event is generated. You can use the VAL-ERROR event in a number of ways. For example, you can set the input focus back onto the field that failed validation. In this case, the $EVENT-DATA register will contain the identity of the entry field that failed validation:

```
VAL-ERROR
    SET-FOCUS $EVENT-DATA
```

To display an error message, you can use a message box. You can either define a message box that displays just that one message, or define it so that it can display different messages by moving the appropriate error message text to the message box.

To display an error message using the second method, you need to define an error message and an error message field in your data definition. The Customer sample screenset uses `ERR-MSG` as the error message field. When a VAL-ERROR event occurs, the following dialog is processed:

```
VAL-ERROR
    SET-FOCUS $EVENT-DATA
    INVOKE-MESSAGE-BOX FIELD-ERROR ERR-MSG $EVENT-DATA
```

The message box `FIELD-ERROR` is used to display the contents of `ERR-MSG`. The $EVENT-DATA register in the third line of dialog is used to contain a code indicating which button the user clicked to acknowledge the message box (for example OK or CANCEL).

3.2.7.1 Validation Criteria

You can use any combination of the following validation criteria:

Range/Table — Where data falls inside or outside a range of specified values including zero. You can specify several ranges as required. The value specified can be numeric or alphanumeric. You cannot use double-byte ranges.

Date — Where data is a valid date in a specified format. You cannot perform date validations on DBCS data items.

Null	Where data may or may not be null (zero or spaces). Nulls are spaces in DBCS, alphabetic or alphanumeric fields, and zeros in numeric fields. You can choose to cause a validation failure if a data item contains all nulls.
User Defined	Where you specify a program name which is called as part of the processing of the VALIDATE function. The program name has a maximum length of 256 characters including its path.

Three data items are validated in the Customer sample screenset:

- C-LIMIT - Range/Table validation with a value of 1000 to 5000.
- C-AREA - Range/Table validation with the values N, S, E, W.
- C-ORD-DT - Date validation with the format DDMMYY.

For more information on the VALIDATE function and the VAL-ERROR event, see the topics *Dialog Statements: Functions* and *Dialog Statements: Events* in the Help.

3.2.8 Error Message Definition

When you specify a data validation, you have the option of associating an error message to be placed in an error message field defined in the Data Block if a validation fails. The way you use error messages will depend on your requirements.

The following rules apply to error message definition:

- All error messages are applied globally to a screenset.
- Only one error message string can be held in the error message field at a time. The next error message overwrites the existing contents of the error message field.
- There can be only one error message file associated with a screenset at one time.

To associate an error message with a validation, highlight the item to be validated in the Data Definition window. Use the **Validation** menu to display the relevant validation dialog box and click **Errors**. The Error Message Definition dialog box is displayed.

The default error message file is **dserror.err**. You can create a new error file, or load an existing error file by selecting **File**.

For the Customer sample screenset, there is an error file **customer.err** already defined. Select this file and click **OK** to return to the Error Message Definition dialog box.

The Customer sample program error file has three error messages:

- 001 The area code must be one of N, S, E or W.

  Use for a validation error on the data item C-AREA.

- 002 The credit limit must lie in the range 1000 - 5000.

  Use for a validation error on the data item C-LIMIT.

- 003 The date must be in valid DD/MM/YY format.

  Use for a validation error on the data item C-ORD-DT.

To specify a new error message, specify a number in **Error number** and the error text in **Error text**, then press **Insert**.

To select an error message to associate with the validation, highlight the message and click **OK**. You return to the validation dialog box. The number of the error message you selected is displayed in **Error message no**.

If validation fails on any of the data input to the data items in the list of error messages just described, a VAL-ERROR event is generated and the appropriate error message is placed in the error message field ERR-MSG.

You can also use the Error Messages dialog box by selecting **Error Messages** on the **Screenset** menu in the main window. This enables you to edit error messages and error message files, but not to associate a message with a validation.

For more information about creating an error message file and defining error messages, see the topic *Data Definition and Validation* in the Help.

3.2.9 Selecting Objects

The next step in the application development process is to select the objects that are appropriate for your application. Objects are the building blocks of Dialog System and you can find them described in the two chapters *Window Objects* and *Control Objects*.

3.2.10 Further Information

Further information on data definitions can be found in the topic *Data Definition and Validation* in the Help, specifically on using data groups, the internal size of data groups, and detailed information on the contents of the generated Data Block.

4 Window Objects

The previous chapters described the basic concepts used in Dialog System and the first step that you need to take to design a graphical user interface. This chapter describes the window objects that are appropriate for your application.

Objects are the building blocks of Dialog System. There are two types of objects, control objects, described in the chapter *Control Objects* and window objects, described in this chapter which covers:

- The components of a window, including its properties and why you would choose particular properties for a window.
- The relationships between primary and secondary windows and some of the effects of the relationship, such as clipping.
- Alternative ways of defining a window.
- How you can handle menu bar options.

A window is an object which is a basic visual element of your user interface. Windows are extremely flexible. They can be sized by the user and have menus in the form of menu bars with pull-down menus. You can put any objects in a window, including other windows.

4.1 Defining Objects for Your Screen Layout

The user interface you build can take many forms, ranging from simple menu selection and data entry fields to fully developed windows complete with menu bars, radio buttons and other controls.

The level of sophistication of the interface you choose depends on many factors including:

- Who the users are and their level of training.
- The hardware and software resources available.

- The complexity of the task to be performed.

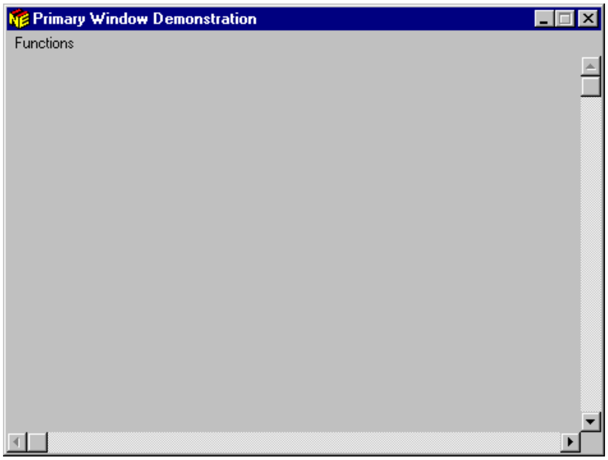
In all cases:

- The user determines the normal course of action.
- The user, rather than your program, is in control of the interface.
- The user interaction with the computer is as simple and natural as possible.

4.2 Components of a Window

Figure 4-1 shows the visual components of a typical window.

Figure 4-1. Typical GUI Window



The properties you choose for your window depend on what you want to display and how you want the user to react to your application. Available properties include:

Title Bar
Area at the top of the window that contains the title of your window. The title of the window in Figure 4-1 is Primary Window.

Minimize/Maximize Icons
Located next to the title bar at the far right of the window, these icons let you quickly minimize or maximize the window or restore the window to its original size.

Border
Defines the extent of the window. The border can be sizable where the user can adjust the size of the window, or non-sizable where the window is fixed in size. The window can also be displayed without a border.

Menu Bar
Located just below the title bar, the menu bar contains the options your application offers to the user. In Figure 4-1, the choice is Functions.

GUIs usually have a pulldown menu, sometimes called an action menu, for each menu bar choice.

Scroll Bars
Provide a visual cue about the position and quantity of information that is visible in the window. Using the scroll bar, the user can adjust the view of the information visible.

The horizontal scroll bar is just above the bottom border of the window and adjusts the horizontal view of the information.

The vertical scroll bar is just to the left of the right border of the window and adjusts the vertical view of the information.

System Menu
Located at the left side of the title bar, the system menu provides menu or keyboard access to the standard window manipulation functions available in your environment.

Client Area
The remainder of the area in the window. This area can contain controls and is where the user performs most of the applications tasks.

4.3 The Desktop

The desktop is the working area on your screen. You can create windows on the desktop. You can have a very large number of windows in your interface (see the topic *Dialog System Limits* in the Help), but only one is displayed when the interface initially runs. This window is called the First Window.

The desktop can contain the following kinds of windows:

- Primary windows.
- Secondary windows.

4.3.1 Primary Windows

The first window you create is called the primary window. It is a child of the desktop (or you could say that the desktop is its parent), because there are no other windows above it. It exists independently of any other window, so any action taken on any other window has no effect on the primary window. You may create more than one primary window.

4.3.2 Secondary Windows

Any primary window can have child windows, called secondary windows. These in turn can have other secondary windows. A secondary (or child) window is used to support the parent window. Many of the actions of a secondary window are determined by the boundaries of the primary window. If you display a secondary window, its associated primary window is also displayed. This relationship between windows has a significant effect on how the user interacts with your application.

For example, a personnel application could have a primary new employee window and multiple secondary windows to enter information such as address, next of kin and other personal details.

If you choose to have a secondary window as the First Window, then both the secondary window and its parent are displayed when the screenset initially runs.

4.3.3 Relationship Between Primary and Secondary Windows

Some of the characteristics of the relationship between primary and secondary windows are:

- If you delete a primary window, all the secondary windows associated with the primary window are also deleted.
- If you move a primary window, all clipped secondary windows move with it and maintain the same relative position on the primary window.
- If you minimize a primary window, all its secondary windows are cleared from the display. When you restore a primary window, all secondary windows are restored to their original position.

Note: The parent of a primary window is the desktop. You can change a primary window to be the child of another window at run time using the function SET-DESKTOP-WINDOW. Refer to the section *Changing the Default Parent Window* in the chapter *Using Dialog*, and the topic *Dialog Statements: Functions* in the Help for a description of this function.

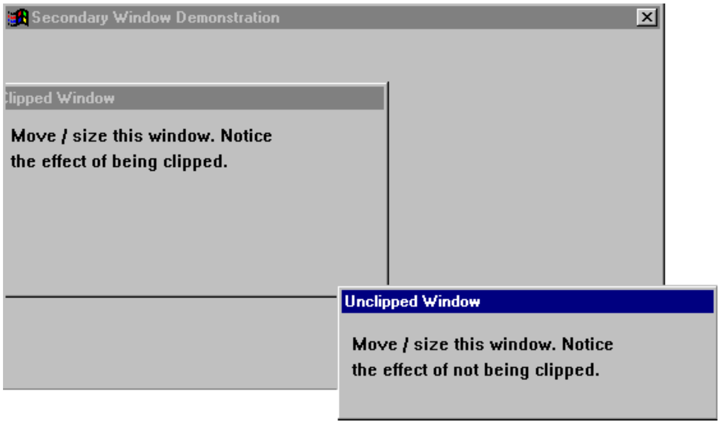
Secondary windows and dialog boxes have similar functions, that is, the display and collection of information. The section *Dialog Boxes Versus Windows* provides some tips on when to use a dialog box and when to use a secondary window.

4.4 Clipping

In Dialog System, you can choose whether or not the secondary window is clipped within the primary window. A window is clipped if the window information is truncated at the border of the parent window.

Figure 4-2 shows an example of a clipped and a non-clipped secondary window.

Figure 4-2. Clipped and Non-clipped Windows



Move / size this window. Notice the effect of being clipped.

Move / size this window. Notice the effect of not being clipped.

You might wish to have a secondary window that is clipped if you need to maintain a particular visual layout of your screen. For example, the window displayed might be similar to a paper form the user is already familiar with, so you can use clipped secondary windows to maintain that visual similarity.

However, a non-clipped window gives users more flexibility. Users can move the secondary windows anywhere on the desktop, and arrange the screen layout however they wish.

The sample application **Objects** shows the effect of both clipping and not clipping a secondary window.

4.5 Defining Windows

You must define a primary window before you can define any other window object. Then you can define either a clipped or a non-clipped secondary window.

Whichever type of window you define, a box appears in the top left-hand corner of the main window. To position and size the window, move the mouse pointer to the position you want the top left-hand corner of the window to be and click to fix this. Move the mouse to the position you want the bottom right-hand corner of the window to be and click once more to fix it. You can then see the new window.

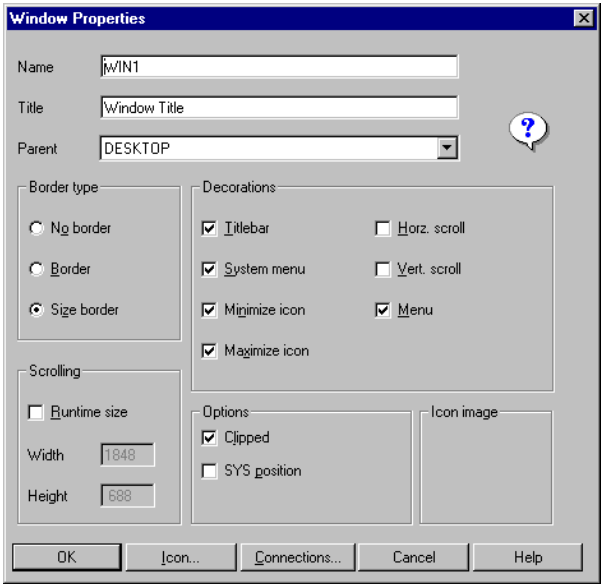
If you specify a dialog box as the child of a clipped child window, its parent will not actually be that clipped child window but the parent of the clipped child window. On creation, the new dialog box is positioned relative to the clipped child window, but thereafter it is positioned relative to the clipped child window's parent. As a result, there is a restriction when painting a screenset in which a clipped window has an unclipped child window. The child window cannot be moved correctly to a new position in the usual way: first you need to change its properties to a clipped window. After the move, the clipped property can be removed.

There are no constraints on the size of a window and it can be larger than the Dialog System main window or positioned outside the main window (if you have Desktop mode set on). When you run the screenset, Dialog System positions and sizes all windows exactly as you created them.

4.5.1 Window Properties Dialog Box

The visual components of a window are described in the section *Components of a Window* earlier in this chapter. The Window Properties dialog box, shown in Figure 4-3, enables you to select the properties for your window.

Figure 4-3. Window Properties Dialog Box



Name: WIN1

Title: Window Title

Parent: DESKTOP

Border type
- ○ No border
- ○ Border
- ● Size border

Decorations
- ☑ Titlebar
- ☑ System menu
- ☑ Minimize icon
- ☑ Maximize icon
- □ Horz. scroll
- □ Vert. scroll
- ☑ Menu

Scrolling
- □ Runtime size
- Width: 1848
- Height: 688

Options
- ☑ Clipped
- □ SYS position

Icon image

See the Help for information on each of these properties

4.5.2 Manipulating a Window

Once you have defined the visual characteristics of a window, you can use dialog to manipulate the window. See the section *Windows Dialog* in the chapter *Using Dialog* for further information.

4.6 Dialog Boxes

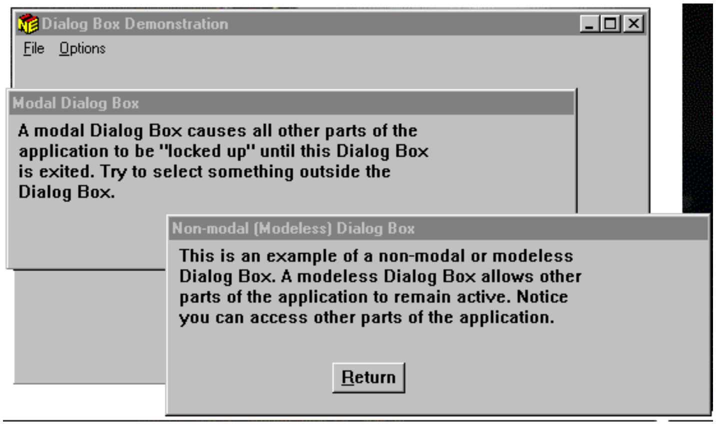
Most often, dialog boxes are used to display or obtain information. This information is displayed or obtained through a collection of controls, such as text fields, data entry fields and buttons, that are displayed on the dialog box. (A maximum of 255 objects can be put on a dialog box.) They have the following properties:

- They cannot be resized or have a menu bar.
- You can add any of the Dialog System control objects to them.
- A dialog box can be created by another window, or by the desktop and is called a child of the window that creates it. The creating window is called the parent of the dialog box.
- Dialog boxes stay in place until they are closed by users' actions.

  Typically, a user clicks a button to cause Dialog System to accept user actions (for example, clicking **OK** to accept a selection from a list). Alternatively, a user may click a button to cause Dialog System to ignore user actions (for example, clicking **Cancel** to close a dialog box, ignoring any input).
- You can manipulate dialog boxes with the same dialog as a window.

Figure 4-4 shows some typical dialog boxes.

Figure 4-4. Typical Dialog Boxes



A modal Dialog Box causes all other parts of the application to be "locked up" until this Dialog Box is exited. Try to select something outside the Dialog Box.

This is an example of a non-modal or modeless Dialog Box. A modeless Dialog Box allows other parts of the application to remain active. Notice you can access other parts of the application.

4.6.1 Modal Versus Modeless

A dialog box can be application modal, parent modal, or modeless. The one you choose depends on how you want the user to react to the dialog box.

A parent modal dialog box means that the window in which the dialog box was created is frozen until the user exits from the dialog box. That is, you cannot access the window until the modal dialog box is exited. However, you can access other primary windows and objects outside the application.

An application modal dialog box means that all other parts of the window hierarchy, and all objects within the application are frozen until the user exits from the dialog box.

You might use an application modal dialog box to ask the user to enter a filename. The application cannot proceed until the user enters a filename (or cancels the action).

Dialog System itself makes use of modal dialog boxes. For example, the About dialog box is modal. (Select **About** on the **Help** menu to see this dialog box.)

A modeless dialog box means that other parts of the application are still active when the dialog box appears. The information that is displayed or requested is not immediately necessary for the continuation of the application.

A modeless dialog box is preferred because it gives the user the maximum flexibility in interacting with the application. However, cases exist, such as the one described above, when an application modal dialog box is necessary.

The sample application **Objects** illustrates application modal versus modeless dialog boxes. Run the screenset to see the effects of each option.

4.6.2 Dialog Boxes Versus Windows

As you can see, windows and dialog boxes are very similar. In fact a dialog box is a special type of window. See the section *Dialog Boxes* in the chapter *The Graphical User Interface*.

Often, you have a choice between using a dialog box or a window in your application. To help you decide when to use a dialog box and when to use a window, we offer the following guidelines:

Use a dialog box instead of a window when:

- The application needs modal entry. For example, if the application requires input from the user and cannot continue until the user responds, use a modal dialog box.
- The amount of information to present or collect is small, does not need a menu bar, and will fit in a dialog box.

Use a window instead of a dialog box when:

- The application requires the user to choose from a list of possible selections from a menu bar.
- The information to present or collect does not fit in the window and the user must resize or scroll the window to view the entire set of data or controls.
- The object needs to have child windows. Using a window gives your application a more logical design.

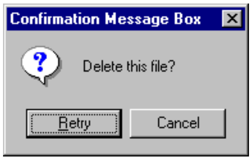
4.7 Message Boxes

A message box is used to give the user a message. It contains only text and/or graphics to present the information, and buttons to let the user choose an action and remove the message box. Usually, these messages are displayed by the application as a result of some event, for example, a "file not found" event.

You cannot define the size or position of a message box.

Figure 4-5 shows a typical message box.

Figure 4-5. Typical Message Box


Confirmation Message Box

Delete this file?

Dialog System provides the following general types of message boxes depending on the nature of the message. You can identify each type by a different icon.

Notification

Notifies the user of a condition to which the user may or may not want to respond. For example, telling the user the printer is out of paper may not affect what the user is currently doing.

Information

Passes information to the user. It requires no response from the user other than confirmation that the message was read. For example, a product information message is an informative message.

Warning

Indicates that a condition exists that may cause undesirable effects. For example, you can issue a warning message when the user is approaching a limit on the number of files opened.

Question

Indicates that a condition exists that requires a user response. For example, the user has selected a **Delete file** option and the message asks the user to confirm the deletion.

Critical

Indicates that a condition exists that will cause undesirable effects if the user continues. For example, the user has opened the maximum number of files and to continue will cause the system to halt.

You must have at least one push button in the message box, so the user can respond to the message. Dialog System provides six pre-defined combinations of push buttons that you can use:

- OK
- OK/Cancel

- Retry/Cancel
- Abort/Retry/Ignore
- Yes/No
- Yes/No/Cancel

For example you could use:

- The **OK** button for all informative messages just to ensure the user received the message.
- The **Yes/No** combination in a message box that requested confirmation that a file be deleted.

You can determine which button the user selected using dialog. The codes that identify the button that was clicked are listed in the description of the INVOKE-MESSAGE-BOX function. See the topic *Dialog Statements: Functions* in the Help.

As an example, suppose the user selects **Delete file** on a menu. The message box (Figure 4-5 above) is invoked so the user can confirm that the file is to be deleted.

4.8 Menus

There are three types of menu:

- Menu bar.
- Pulldown menu.
- Context menu.

4.8.1 The Menu Bar

The menu bar is a list of commands at the top of the window - or more accurately, it is a list of groups of commands, because selecting one command usually presents the user with further choices.

See the topic *Objects and Properties* in the Help for information on defining a menu bar.

Menu choices can:

- Pull down additional menus. These are identified by arrows following the choice.

  See the section *Pulldown Menus* below.

- Cause a secondary window or a dialog box to be displayed. These are identified by an ellipsis (...) following the choice.

- Be check-marked. These have two states, on or off. When you click on the choice, you reverse its state. If it is on, a checkmark appears in front of it. These choices are sometimes called toggles.

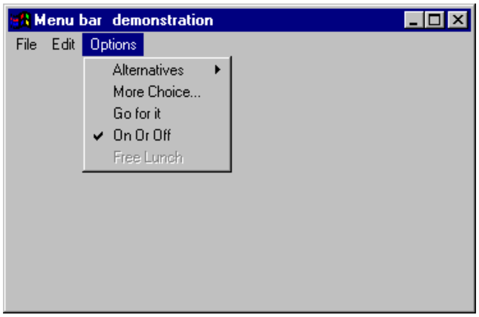
- Cause an action to be performed directly. These have no special identification.

Any one menu choice can only have one of the actions described in the preceding list, and is decorated (for example with an ellipsis or a checkmark) appropriately.

4.8.2 Pulldown Menus

When you select a choice on the menu bar, a pulldown menu drops down from the choice. These are the actions you can perform on the object. Figure 4-6 illustrates a typical pulldown menu.

Figure 4-6. Typical Pulldown Menu



There are several possible actions that can happen after you select a choice. Usually, the menu choice includes a visual cue to indicate the type of action. The following list describes visual cues and their associated actions.

Arrow	Invokes another pulldown menu at the side of the current one to give you further menu alternatives from which you can choose.
Checkmark	Shows whether or not a choice is in effect. When the checkmark is toggled on, the choice is operational. When it is off, the choice is not operational. When a checkmark choice is off, you cannot distinguish it from an ordinary menu choice that performs immediately.
Dimmed	Shows that the choice is not available.
Ellipsis (...)	Invokes a dialog box or secondary window presenting a more complex choice, for example choosing from a list of files or selecting a radio button.
No distinguishing marks	Performs the action immediately.

Examples of menu bar dialog are discussed in the sections *Menu Bar Dialog* and *Enabling and Disabling Choices* in the chapter *Using Dialog*.

The topic *Objects and Properties* in the Help tells you how to define menu bars.

4.8.3 Context Menus

Context menus provide a quick and easy method for performing operations relevant to your work area in Dialog System. If you right-click on the screen, Dialog System displays a menu of choices appropriate for the item you have clicked on. For example, the context menu is different for a push button object in the main definition window from a dialog line in the dialog definition window.

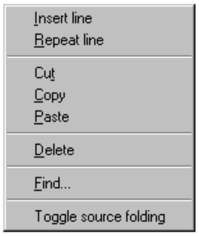
An example of a context menu is given below:

Select **Global dialog** on the **Screenset** menu.

Right-click on any of the lines of dialog.

Dialog System displays a context-sensitive menu, as shown in Figure 4-7.

Figure 4-7. Global Dialog Context Menu



4.9 Attaching an Icon

An icon is a small graphic that represents a window when it is minimized. When you create a window or dialog box, you can attach an appropriate icon to use when it is minimized.

To attach an icon to a window:

1 Select the window.

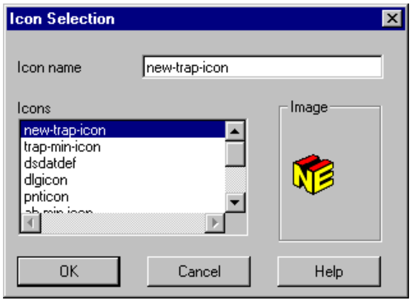
2 Select **Properties** on the **Edit** menu.

The Window Properties dialog box is displayed.

3 Click **Icon**.

This displays the Icon Selection dialog box, shown in Figure 4-8, which contains the list of icon names available in the bitmap file. The file **ds.icn** is always the default.

Figure 4-8. Icon Selection Dialog Box



Icon name: new-trap-icon

Icons: new-trap-icon, trap-min-icon, dsdatdef, dlgicon, pnticon

4 Select an icon name from the list (its image will be shown).

5 Click **OK** to return to the Window Properties dialog box.

Ensure that Minimize Icon is selected or the icon will not be used.

6 Click **OK** to return to the Object Definition Window.

5 Control Objects

The previous chapter looked at window objects. This chapter describes:

- Control objects and their use.
- Grouping controls.
- Aligning controls.

5.1 Control Objects

Control objects are also known as widgets or gadgets and they can be used in windows and dialog boxes. They cannot exist outside a window or dialog box, so you have to define a window or dialog box and select it before you can define controls. See the chapter *Window Objects* for further details.

Depending upon the control, you can use it to:

- Input data (for example, by presenting alternative values for the user to select).
- Navigate among different displays.
- Display data.

The controls covered in this chapter are:

Control	**Use to**
Text and Entry fields	Read and enter data items, and supply text for labeling.
Push buttons	Enable the user to select an immediate action.
Radio buttons	Give the user a fixed set of mutually exclusive choices, when grouped.

Control	Use to
Check boxes	Give the user a set of choices that are not mutually exclusive, that is, the user can select none, any, or all of the choices.
List boxes	Display a list of choices from which the user can select one or more.
Selection boxes	Present a scrollable list of choices and place a selection in an entry field.
Scroll bars	Enable the user to scroll to a particular value, when used with another control like an entry field.
Group boxes	Outline a control or group of controls for the visual convenience of the user.
Tab control and Tab control pages	Organize information like a spiral bound notebook containing pages and tabbed divider pages.
OLE2 controls	Display a window from which the user can call any external object, for example, a spreadsheet.
User controls	Contain objects not paintable by the Dialog System definition software. How you use a user control depends on its implementation.
ActiveX controls	Display objects with predefined functions, usually supplied by third parties.

5.1.1 Text and Entry Fields

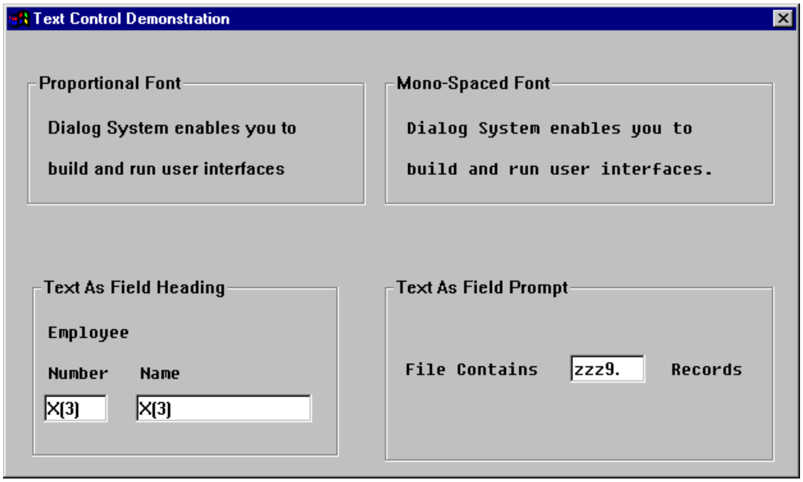
The most basic controls are entry fields. This section describes:

- Some of the characteristics of text, single line and multiple line entry fields.
- Using entry fields with other controls.

5.1.1.1 Displaying Text (Text Objects)

A text object is a control that you can use to help your user understand what information is displayed or asked for. You can think of a text field as a label. Figure 5-1 shows an example of the use of text fields.

Figure 5-1. Examples of Text Fields



Dialog System enables you to build and run user interfaces

Dialog System enables you to build and run user interfaces.

X(3) X(3)

File Contains zzz9. Records

You can use text fields as:

- Field or column headings.
- Field prompts.
- Descriptive text.

A text field is static. That is, no events or functions are associated with a text control. If you need a text field that you can change at run time, such as a status line or a dynamic prompt, use a display-only entry field. See the section *Display-only Entry Fields* for more information.

The topic *Objects and Properties* in the Help provides detailed information on defining text fields.

5.1.1.2 Getting Input Using Entry Fields

Entry fields are control objects that display the contents of a data item at run time and enable the user to edit the contents. You use an entry field when you do not know the possible values for a data item. In this situation, the user must supply the needed information.

An entry field is associated with a data item. Changes to the entry field made by the user are immediately put into its data item with no further action on your part. If the contents of the data item change, the entry field must be refreshed before it reflects the change. An entry field is refreshed when:

- Its parent window is created at run time.
- It, or its parent window, is refreshed explicitly by the dialog function REFRESH-OBJECT.
- It receives the input focus.

An entry field is not refreshed when its parent window is shown using the SHOW-WINDOW function.

If you set focus on a numeric entry field and the associated master field does not contain valid numeric data, the value will be set to zero.

Dialog System provides two types of entry fields - single line entry and multiple line entry (MLE).

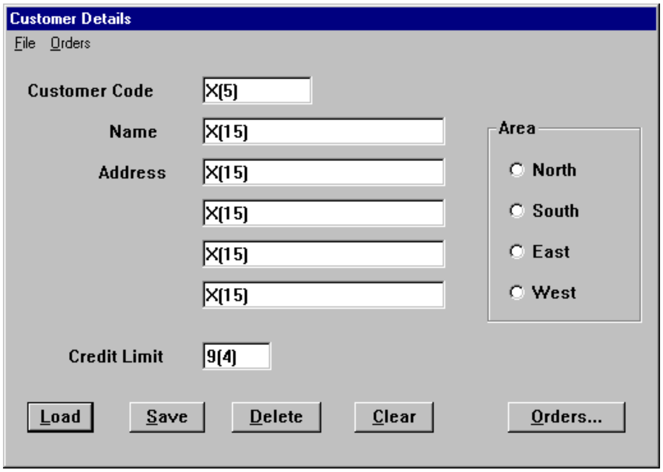
5.1.1.2.1 Single Line Entry Field

The simplest entry field is a single line entry field, which can display or collect only one line of text.

As an example, in the Customer sample, users must supply information such as customer code, name, address and credit limit. They enter this information using entry fields.

Figure 5-2 shows several examples of single line entry fields.

Figure 5-2. Examples of Entry Fields



Customer Code X(5)

Name X(15)

Address X(15)

X(15)

X(15)

X(15)

Credit Limit 9(4)

Area

○ North

○ South

○ East

○ West

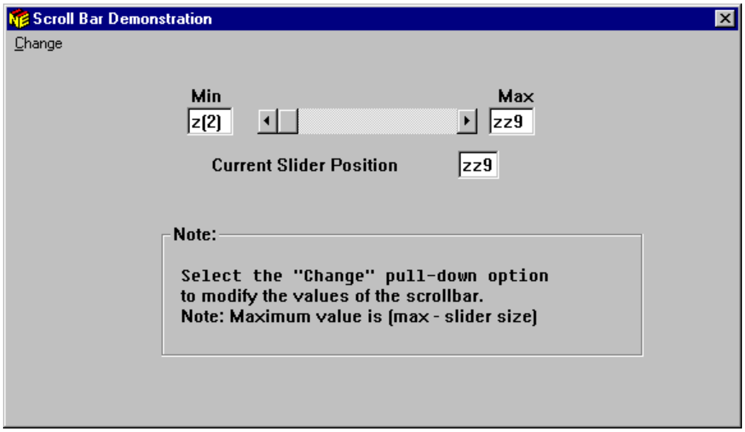
5.1.1.2.2 Using Entry Fields with Other Controls

You can also use entry fields with other controls to help the user select appropriate values.

For example, if you know the data item is in a particular range of values, you could use the entry field with a scroll bar to let the user "scroll" to the correct value.

Figure 5-3 shows an example of using an entry field with a scroll bar.

Figure 5-3. Using an Entry Field with Scroll Bar



Select the "Change" pull-down option
to modify the values of the scrollbar.
Note: Maximum value is (max - slider size)

You can find an example of the dialog needed to implement this feature in the chapter *Sample Programs*.

The topic *Data Definition and Validation* in the Help describes how to set up validation checks using Data Definition.

5.1.1.2.3 Display-only Entry Fields

Some entry fields need to be display-only. For example, you might have a dialog box that lets the user update address information for employees. You might want the user to see, but not update, some fields, for example the employee name.

Another use of display-only entry fields is to build dynamic prompts or status lines. For example, you might not know the exact wording of a prompt at definition time. You can make the prompt a display-only entry field, then supply the correct text at run time.

To illustrate how to make a dynamic prompt, you might have a display-only entry field that is associated with an item in the Data Block named `generic-prompt`. In your program, fill the item using a statement like:

```
    move "Dynamic Prompt" to generic-prompt
```

The new prompt is then displayed when the entry field associated with `generic-prompt` is refreshed.

Display-only is a property of single line entry fields. The topic *Objects and Properties* in the Help describes how you set an entry field to display-only.

5.1.1.2.4 Autoswipe

Autoswipe is a property of an entry field: when the user tabs to the field, all the data associated with that field is selected. If the user inputs any data in the field, the field is cleared before the new data is entered.

You can use Autoswipe to clear a default value when you know the user would not use the default. For example, if you use an entry field to enter a filename, you may want the default to be a descriptive string like **filename.ext**. This gives the user an indication of the format you expect for the filename, but it is unlikely that a file would have that default name.

As soon as the user enters any data in the field, the existing text is cleared and the data entry begins in the first position of the field.

The SET-AUTOSWIPE-STATE function lets you change the Autoswipe state of an entry field. In the example above, once the user has entered a valid filename, you may want to set SET-AUTOSWIPE-STATE off. Then when the user tabs to this field, the field is not cleared before data is entered. See the topic *Dialog Statements: Function* in the Help.

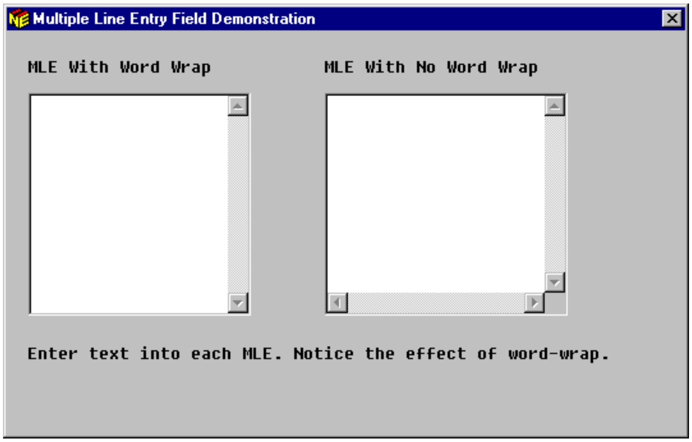
5.1.1.3 Multiple Line Entry Fields

Like a single line entry field, a multiple line entry field (MLE) enables you to display and edit the contents of a data item. However, this control is most useful when you have a lot of information to collect, for example a lengthy text description of a product.

The MLE appears on the screen as a box with horizontal and vertical scroll bars. The scroll bars allow you to adjust the view of the data in the MLE. The scroll bars do not have any dialog associated with them; that is, Dialog System controls all the scroll bar actions for the MLE.

Figure 5-4 shows two MLEs. The one on the left has the word-wrap option set and the one on the right does not.

Figure 5-4. Multiple Line Entry Field



MLE With Word Wrap

MLE With No Word Wrap

Enter text into each MLE. Notice the effect of word-wrap.

The data item associated with the MLE can contain alphanumeric and DBCS characters, and line feeds (ANSI decimal 10, x"0A"), which place successive characters on the next line in the display area of the MLE as you would expect. The user can view characters that extend beyond the right-hand or lower boundary by using the scroll bars.

You can set the MLE to word-wrap the displayed text so it never extends beyond the right-hand boundary. Single words that cannot fit are simply split.

The **Objects** sample program shows the differences between an MLE that has the word-wrap option set and an MLE that does not.

5.1.1.4 Editing an MLE

You can put or edit text in the MLE by:

- Moving the text to the associated data item in your application program.
- Moving the text to the data item using dialog. In this case, line feeds cannot be inserted.

- Loading text directly to the MLE from the Clipboard .
- Allowing the user to edit the information directly in the MLE.

5.1.1.5 Refreshing an MLE

When an MLE's data item is changed at run time, the MLE needs to be refreshed before the change is visible. An MLE is refreshed when:

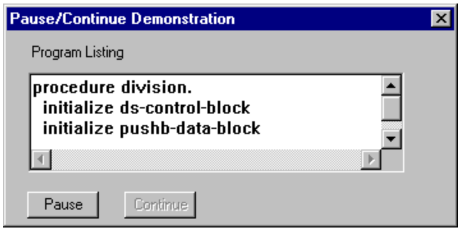
- Its parent window is created at run time.
- It or its parent window is refreshed explicitly by the dialog function REFRESH-OBJECT.

Changes made by the user to the MLE are immediately put into its data item with no further action on your part.

5.1.2 Push Buttons

A push button lets a user select an action. The action indicated by the text in the push button occurs immediately. Figure 5-5 shows an example of a window with a push button.

Figure 5-5. Typical Push Button



```
procedure division.
  initialize ds-control-block
  initialize pushb-data-block
```

5.1.2.1 Assigning Bitmaps to Push Buttons

When you define a push button, you can use the Push Button Properties window to attach a set of bitmaps, instead of text, to a push button. You define a bitmap for each of the three possible states a push button can be in; normal, disabled, and depressed.

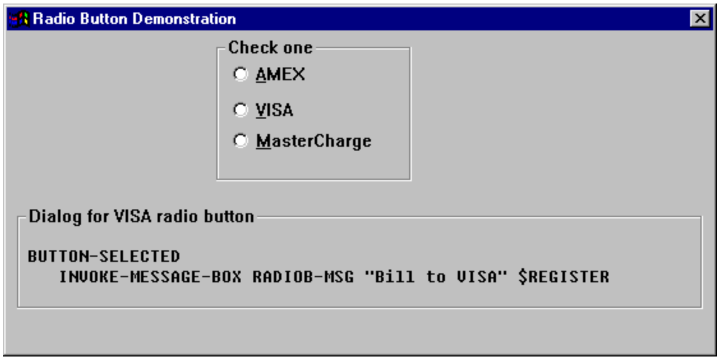
You must define a bitmap for the normal state. If you do not define a bitmap for the depressed or disabled states, the bitmap defined for the normal state is used.

However, you might want to set or change bitmaps at run time. See the chapter *Using Bitmaps to Change the Mouse Pointer* for further information.

5.1.3 Radio Buttons

Unlike push buttons, you can group radio buttons together to show the user a fixed set of mutually exclusive choices. For example, Figure 5-6 shows this use of radio buttons, in which the user must select only one method of billing.

Figure 5-6. Typical Radio Buttons



Dialog for VISA radio button

```
BUTTON-SELECTED
   INVOKE-MESSAGE-BOX RADIOB-MSG "Bill to VISA" $REGISTER
```

Grouping radio buttons gives them the characteristic behavior, which is that only one button in the group can be selected at a time. When the user selects a new button, the previously selected button is deselected.

See the section *Grouping Controls* in this chapter for a description of how to define a control group.

The dialog associated with a radio button is the same as the dialog for a push button. For example, the radio button VISA can have dialog such as:

```
BUTTON-SELECTED
    BRANCH-TO-PROCEDURE BILL-TO-VISA
```

where the procedure `BILL-TO-VISA` contains the necessary billing functions.

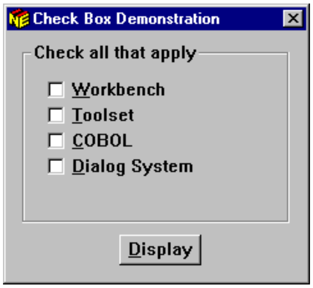
Note: You can deselect a radio button only by selecting another radio button within the control group.

For information on defining a radio button, see the topic *Objects and Properties* in the Help.

5.1.4 Check Boxes

Check boxes are much like radio buttons except the user can select choices that are not mutually exclusive. For example, as shown in Figure 5-7 you can use check boxes when you have a "check all that apply" situation.

Figure 5-7. Group of Check Boxes



You can attach a numeric data item to a check box. If the check box is set, the data item associated with the check box is set to one. If the check box is not set, the data item is set to zero.

See the chapter *Sample Programs* for further information.

5.1.5 List Boxes

You use list boxes to display a list of choices from which the user can select one or more. A list box consists of:

- A window containing the list of items.
- At least one scroll bar to allow you to move unseen information into view.

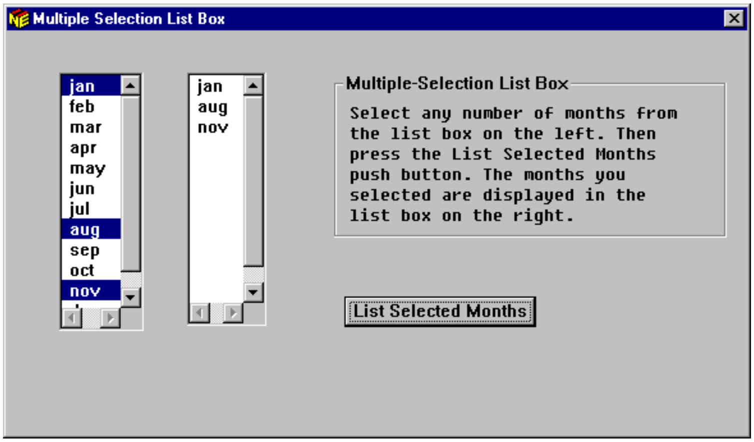
Dialog System provides three types of list boxes:

- Single selection - Select only one entry from the list.
- Multiple selection - Select any number of entries (up to the limit of the list box).
- Extended selection - Select any number of adjacent entries (up to the limits of the list box).

The **Objects** sample application shows the differences between a single selection list box and a multiple selection list box.

Figure 5-8 shows an example of a list box.

Figure 5-8. Example of a List Box



Select any number of months from the list box on the left. Then press the List Selected Months push button. The months you selected are displayed in the list box on the right.

For information on defining a list box, see the topic *Objects and Properties* in the Help.

5.1.5.1 Adding Items to a List Box

You can insert items into the list box in the following ways:

- Using a group item passed from your program.

  You can associate the list box with a group item, such that each line in the list box displays an occurrence of the group. Selected items in the group then occupy fields in each line.

  This is probably the most common way of using a list box. For an example program, see the chapter *Sample Programs*.

- Using the definition facility when you define the list box properties.

  You can type in items at definition time as part of the list box properties . These items are inserted in the list when the box is created at run time. Subsequently, they are just like any other inserted list items. For example they can be changed, deleted or have other items inserted between them.

This is an effective way of adding items to the list if you have a small number of entries that you always want to have in the list. See the topic *Objects and Properties* in the Help for further information on how to add choices to a list box in this way.

- Using dialog at run time using the INSERT-LIST-ITEM function.

  This is an effective way of populating a list box if you have a few choices in your list and you want to keep the Data Block as small as possible. Although it does not require much time to fill a small list box, it does take a little time and adds to the number of dialog statements that have to be executed.

- Passing a delimited string from your program.

  This method offers two advantages:

  - It reduces the number of dialog statements in your screenset.
  - It enables you to maintain the list outside Dialog System. For example, you could have the data stored in an ASCII file that you can maintain with an editor and read in with your program at run time. Thus, you can make minor changes to the data without affecting either your program or the screenset.

  It does, however, add to the size of the Data Block passed between your program and Dialog System, but it is faster than adding the data items to the list one at a time.

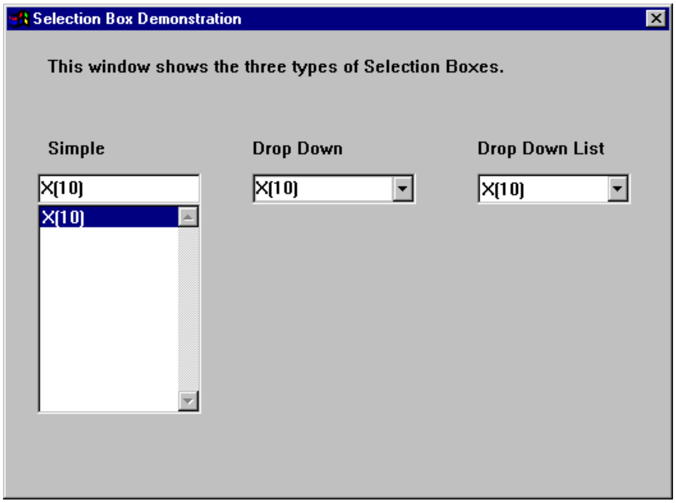
  For an example, see the chapter *Sample Programs*.

5.1.6 Selection Boxes

A selection box is a control object that is used to present a scrollable list of choices and place a selection in an entry field. It is sometimes known as a combination box or combo box.

Figure 5-9 shows the three types of selection boxes that are available.

Figure 5-9. Typical Selection Boxes



Selection Box Demonstration

This window shows the three types of Selection Boxes.

The following describes the types of selection boxes:

Simple (or fixed) Consists of an entry field and a list box. You can either type in the entry field, or select an item from the list, which is then placed in the entry field.

Drop-down Consists of an entry field with an arrow button at the right-hand end and a hidden list box. When you press the arrow button, the list box drops down. The user can type in the entry field, or select an item from the drop-down list to be placed in the entry field. The list box disappears again when the user makes a selection or clicks elsewhere.

Drop-down list Consists of a display-only entry field with an arrow button at the right-hand end and a hidden list box. You cannot type into the display-only field; you must select a choice from the list. You can think of it as a list box that shrinks to show only the current selection.

You define the three types of selection box in exactly the same way, then choose the specific type you want as a property in the Selection Box Properties dialog box.

To add selection boxes individually to windows or dialog boxes:

1. Select the window or dialog box.
2. Click **Selection box** on the **Objects** toolbar
   - or -
   Click **Selection box** on the **Object** menu.

If you define a selection box where the drop-down list will obscure other controls in the window or dialog box, ensure that the selection box receives focus before the controls that might be obscured. Either define the selection box first, or use **Controls** on the **Edit** menu to change the control order. This ensures that all the controls are properly displayed at run time.

5.1.6.1 Entry Field

The entry field part of the selection box is linked to a data item called its master field. You normally place text in the entry field just to supply a default choice.

5.1.6.1.1 Refreshing the Entry Field

To display the contents of the master field in the entry field at run time, the selection box must be refreshed. A selection box is refreshed when:

- Its parent window is created at run time.
- It or its parent window is refreshed explicitly by the dialog function REFRESH-OBJECT.

If you show a selection box using the SHOW-WINDOW function, it is not refreshed.

5.1.6.1.2 Changing the Entry Field

If the user makes changes to the entry field by typing or selecting a choice in the list box, these changes are immediately put into the master field with no further action on your part.

Selecting an item in a selection box causes an ITEM-SELECTED event to be generated.

You can place text items in the list box in the following ways:

- Type in items at definition time as part of the selection box properties.

  These items are inserted in the list when the selection box is created. After that, they are just like any other list item. For example, they can be deleted or have other items inserted between them.

- Insert and delete items at run time using dialog functions INSERT-LIST-ITEM or INSERT-MANY-LIST-ITEMS and DELETE-LIST-ITEM.

  For a description of this (and other) methods of adding items to the list box see the section *Adding Items Using Dialog* in the chapter *Sample Programs*.

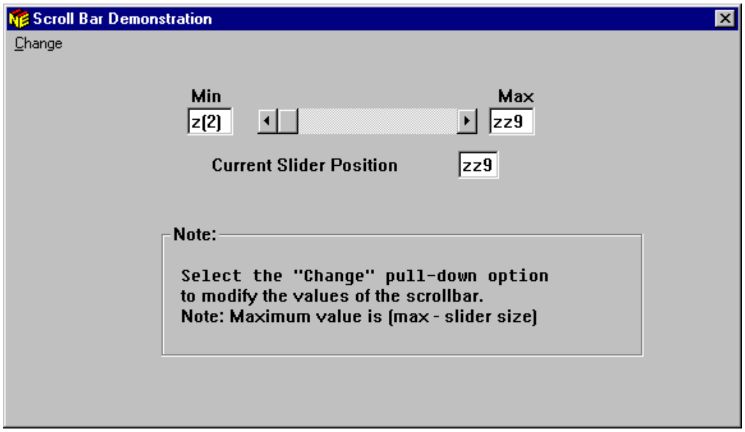
5.1.7 Scroll Bars

In a list box and an MLE, a scroll bar indicates the position and quantity of information visible in the list box. Using the scroll bar, the user can adjust the view of the information visible. However, in Dialog System you can also create a scroll bar as an independent control.

As an example, you can use the scroll bar with an entry field to let the user "scroll" to the correct value. The scroll bar example in the sample application **Objects** shows this usage.

Figure 5-10 shows a horizontal scroll bar.

Figure 5-10. Horizontal Scroll Bar



Select the "Change" pull-down option
to modify the values of the scrollbar.
Note: Maximum value is (max - slider size)

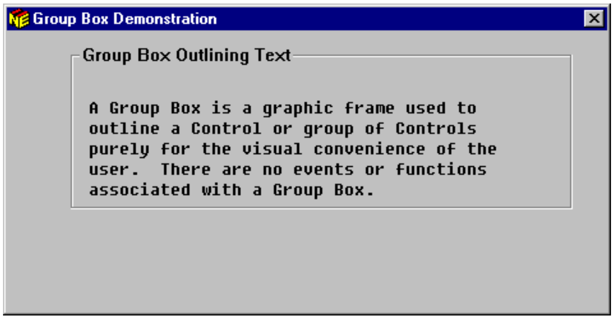
The Scroll Bar Properties dialog box lets you assign defaults to scroll bar properties such as slider range, slider position and slider size. Using dialog, you can change these properties.

You can see a coded example in the chapters *Tutorial - Creating a Sample Screenset* and *Tutorial - Using the Sample Screenset*. For information on defining a scroll bar, see the topic *Objects and Properties* in the Help.

5.1.8 Group Boxes

A group box is a graphic frame used to outline a control or group of controls purely for the visual convenience of the user. This encourages the use of several controls to be understood as a group. Figure 5-11 shows an example of a group box.

Figure 5-11. Typical Group Box



Group Box Outlining Text

A Group Box is a graphic frame used to outline a Control or group of Controls purely for the visual convenience of the user. There are no events or functions associated with a Group Box.

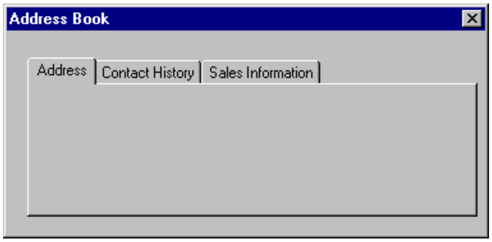
Like a text control, no events or functions are associated with a group box.

For information on defining a group box, see the topic *Objects and Properties* in the Help.

5.1.9 Tab Controls

A tab control enables related information to be organized in a way that is intuitive and easy to use. Figure 5-12 shows an example of a tab control.

Figure 5-12. Typical Tab Control



A tab control is made up of individual pages. These are combined into a single control with each page represented as a tab along one edge (usually the top) of the control. Selecting a tab displays the associated page.

Tabs can contain text or bitmaps. If all the tabs do not fit on a single line, they can be displayed either with scroll arrows to move through the tabs, or the tabs can be wrapped over several lines so that they are all visible.

The visible area of the tab control is always the top page. Pages can contain any Dialog System control objects, apart from another tab control or window.

You can see a coded example in the chapter *Sample Programs*.

5.1.10 OLE2 Controls

An OLE2 control is a window, inside which you can display external data or objects, such as spreadsheets and bitmaps.

As an example, you can use the Paintbrush application to design and edit a picture, and then load that picture into your OLE2 window.

5.1.11 User Controls

You use a user control as a container for an object that is not paintable by the Dialog System definition software.

Events and functions on a User Control are implemented in a controlling program associated with the object. Typically the controlling program uses the class library to create and maintain the object.

You create and manipulate User Controls by generating the controlling program at definition time. The advantage of using a controlling program is that the object is completely user-definable while still having the benefits of being integrated with the Dialog System and run-time software.

Integrating user control objects with the definition software provides the following benefits:

- Automatic object creation and display at run time.

- Depending on the type of object, support for screens of differing resolutions, as well as the ability to resize and move child windows and gadgets. For more information see the chapter *Advanced Topics* and the topic *Dynamic Window Sizing* in the Help.
- Position and size of objects can be easily defined and modified in the same way as other Dialog System objects.
- Objects can be reordered in the same way as other Dialog System objects. This allows the tabbing order at run time to be defined. The user can define the sequence in which they tab between existing Dialog System objects and user control objects.
- All standard alignment functions are available on user controls, enabling alignment with other Dialog System objects. For information on aligning objects see the section *Aligning Controls* in this chapter.

For information on creating and manipulating user control objects, see the chapter *Programming Your Own Controls*.

5.1.12 ActiveX Controls

ActiveX controls are supplied by third-party vendors and can be integrated into Dialog System applications. The benefits listed above for user controls also apply to the use of ActiveX control with your screenset. Additional benefits of using ActiveX controls are that you can:

- Paint them visually at definition time.
- Display the control's property pages dialog for storing initial properties in the screenset.
- Get help on writing code to manipulate the run-time appearance and behavior of the control via properties, methods and events, using the Dialog System Programming Assistant. See the chapter *Programming Your Own Controls* for further details.

You need to purchase a development licence for any controls you may wish to use in your application. You are also bound by the vendor's licence agreement for the production and distribution of that control.

5.2 Grouping Controls

When controls, such as entry fields, radio buttons, and push buttons appear in a window or dialog box at run time, the user can move the keyboard input focus around them by pressing **Tab**.

Sometimes, several controls form a logical group (for example, a set of radio buttons) and either the user is interested in all the controls in the group or none of them. Therefore, it is convenient for successive presses of **Tab** to move the focus onto the group and then away to the next control or group. You can use the cursor keys to move around controls in a group. (Again, this is the default action, which you can change if you choose.)

You can achieve this behavior by putting the controls into a control group.

You must put a group of radio buttons into a control group so that they have the characteristic behavior that only one radio button in the group can be selected at a time. When one is selected, all the others in the same group are unselected.

Usually, you want the focus to start on the control or control group that the user is most likely to operate, and move around them in a convenient order with successive presses of **Tab**. The order of controls within each group also needs to be specified so the focus lands on the first control when **Tab** is used, and the cursor keys move the focus around the group's controls in a convenient order.

Note: You cannot **Tab** into a control group when the first item of the group is disabled.

You can specify the order of controls and groups in each window or dialog box. The procedure for doing this is explained in the topic *Dialog System Overview* in the Help.

5.3 Aligning Controls

Dialog System provides an alignment mechanism that makes it easy to align graphical objects when moving or sizing in a graphical environment.

Most of the alignment functions are available on the Alignment toolbar, which is available on the Alignment tab page. See the topic *Dialog System Overview* in the Help for a description of the Alignment toolbar.

The alignment options mean that you don't have to position controls exactly when you define them. You can quickly add the controls you need in rough form and then use the Alignment toolbar buttons to set the spacing and sizing.

Most alignment operations will require a combination of the alignment functions. For example, you might want to place a set of four push buttons near the bottom of the window. The steps you might follow are:

1. Position the push buttons roughly in the area you want them to appear.
2. Choose **Select area** on the **Edit** menu or use the select area toolbar option to enclose the objects. This enables the objects to be treated as a group.
3. Select the **Align to bottom** button.
4. Select the **Equalize objects' width** button.
5. Select the **Equalize objects' height** button.

If the spacing is not ideal, there is an alternative way to align the buttons:

1. Select **Tile horizontally.**

   This effectively gives no spacing.

2. Size the group by:

   - Selecting **Size** on the **Edit** menu.

     -or-

- Using the sizing handles on the selected controls.

  This increases the spacing between the controls.

Suppose now that the four push buttons are not in the correct order. You can rearrange them by using the **Shuffle** button. For example, the four push buttons might be labelled **OK**, **Insert**, **Cancel** and **Help** respectively and you want to change the order to **Help**, **Cancel**, **Insert**, and **OK**.

To rearrange the buttons, first select them in the order you want them. To do this:

1. Click on the **Help** push button.
2. Hold down the **Ctrl** key and click on the **Cancel** push button.

   Repeat this step for the **Insert** and **OK** push buttons.
3. Click **Reorder objects** on the Alignment toolbar.

   The order becomes **Help**, **Cancel**, **Insert** and **OK**.

Each button has been swapped with another in the group into the order in which you selected them. The default order of swapping is left to right. That is, in this example, the **OK** button is the first to swap, **Insert** the second, **Cancel** the third and **Help** the fourth.

With all the alignment functions, you can undo the last operation by selecting **Undo** on the **Edit** menu.

The following steps describe how to space the four push buttons vertically down the right-hand side of the window:

1. Select the controls as a group in the same way as described previously.
2. Move the group towards the top of the window.
3. Click **Tile vertically**.
4. Click **Align to right**.
5. Select **Size** on the **Edit** menu to stretch the spacing between the controls.

   You can now move the controls to their ideal position on the window and use **Reorder objects** to rearrange the order if necessary.

5.4 Sample Program

You can see examples of the Dialog System Control objects by running the **Objects** sample program supplied in your installation **DialogSystem\demo\objects** directory.

5.5 Using Bitmaps

This section explains how to use bitmap graphics with your Dialog System interface. Dialog System enables you to choose the bitmaps you want to use in your user interface for the following objects:

Bitmap object	A decorated object the user can select.
Icon	A small bitmap that represents the window or dialog box when it is minimized. See the section *Attaching an Icon* in the chapter *Window Objects*.
Mouse pointer	A bitmap is used to indicate the mouse pointer, and indicate the different states of the mouse or the system. See the chapter *Tutorial - Using Bitmaps to Change the Mouse Pointer*.
Push buttons	A bitmap is used to decorate the button or show the button in different states.

Dialog System includes bitmaps for the objects it uses and some standard bitmaps for the environment you are working under. You can also add your own bitmaps, icons and mouse pointers to use in a screenset. For more information, see the topic *Bitmaps, Icons and Mouse Pointers* in the Help.

5.5.1 Defining Bitmaps

Bitmaps can be placed anywhere within a window or dialog box. Their purpose is mainly for decoration, to draw attention to buttons or to place items such as logos in a window or dialog box. A mouse click over a bitmap causes a BITMAP-EVENT to occur, which places the bitmap's object handle in the $EVENT-DATA register. You can also define dialog to react to the MOUSE-OVER event for a bitmap.

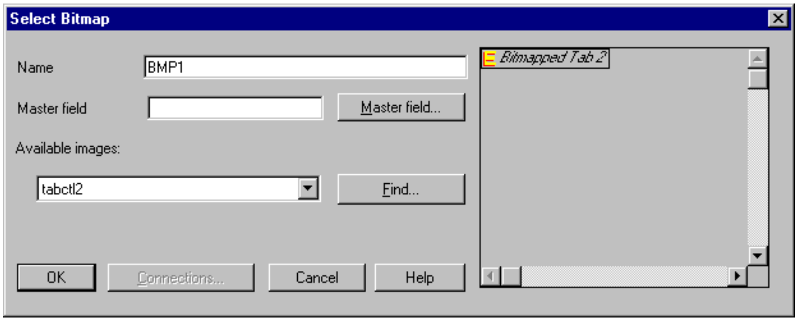
You can assign a Master field to a bitmap from its Properties dialog box. This enables the displayed bitmap image to be changed dynamically at run time. For more information on this, see the topics *Using Controls* and *Bitmaps, Icons and Mouse Pointers* in the Help.

To place a bitmap in a window or dialog box:

1 Select **Bitmap** on the **Object** menu in the main window.
-or-
Click the bitmap icon in the **Objects** toolbar.

The Select Bitmap dialog box shown in Figure 5-13 is displayed.

Figure 5-13. Select Bitmap Dialog Box



Select Bitmap

Name: BMP1

Master field:

Available images: tabctl2

2 Specify the bitmap name in **Name**
-or-
Select it from **Available images**.

3 Click **OK** when you have made a choice.

The Select Bitmap dialog box disappears and the outline shape of the bitmap appears in the window or dialog box.

4 Use the mouse to position the bitmap and click.

The bitmap now appears.

You can use **Controls** on the **Edit** menu to put other controls after the bitmap in the list. **Controls** has no effect on the display until the window is hidden and reshown.

5.6 Bitmapped Push Buttons

Dialog System provides default appearances for push buttons. You can use bitmapped push buttons to change the appearance of a push button.

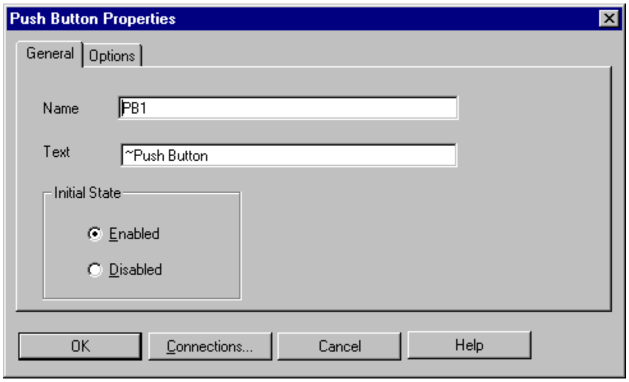
For example, when you want a button to convey a graphic symbol, or you want to make the push button appearance vary depending on its state.

To associate bitmaps with a button:

1 Double-click on the button.
-or-
Select the button then choose **Properties** on the **Edit** menu in the main window.

The Push Button Properties dialog box, shown in Figure 5-14 is displayed.

Figure 5-14. Push Button Properties Dialog Box



Push Button Properties

Name: PB1

Text: ~Push Button

Initial State

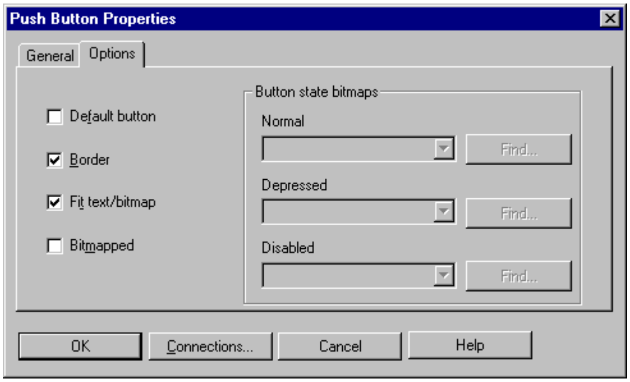
● Enabled

○ Disabled

2 Click the **Options** tab.

You see the options available, as in Figure 5-15.

Figure 5-15. Push Button Properties Dialog Box - Options



Push Button Properties

☐ Default button

☑ Border

☑ Fit text/bitmap

☐ Bitmapped

3 Select **Bitmapped**.

This option becomes marked and the Button State Bitmaps options become available. There are three choices:

- Normal
- Depressed
- Disabled

4 Select the bitmap file you require from the Button state bitmap dropdown lists.

This bitmap file name is displayed as the selected bitmap file.

5 Click **OK** to return to the main window.

When creating a window toolbar, you may want to tailor your own version of the toolbar control program to implement a native Win32 toolbar.

6 Using Dialog

The preceding chapters have described the visual aspects of the screenset. Now we are going to look at how to activate the interface using dialog. In this chapter you will see how powerful dialog is, by looking at:

- Dialog and its levels.
- Events, functions and procedures.
- Special registers.
- Important examples of dialog events and functions.

6.1 What is Dialog?

Dialog is a language which works behind the scenes to trap events and perform the actions you want to happen when the event occurs. A simple event and response could be, for example, "if the user presses a button labelled **Next window**, set the focus on a new window". You specify the code to perform this using dialog which is accessed by your application program.

A piece of dialog consists of:

- The names of events that can occur and to which you want to respond.
- For each event, instructions (called functions) to perform when the event occurs.
- Optionally, procedures (named subroutines) that contain a list of functions that can be called.
- Optionally, comments to make the dialog more readable.

An event or a procedure name ends the dialog for the preceding event. For example, consider the following dialog fragment attached to a window.

```
CLOSED-WINDOW
    SET-EXIT-FLAG
    RETC
F1
    MOVE 1 ACTION
    RETC
    SET-FOCUS EMPLOYEE-WIN
  DELETE-EMPLOYEE-PROC
    MOVE 2 ACTION
    RETC
    SET-FOCUS EMPLOYEE-WIN
```

The functions SET-EXIT-FLAG and RETC are attached to the CLOSED-WINDOW event. That is, RETC is the last function associated with the CLOSED-WINDOW event. The functions MOVE 1 ACTION, RETC and SET-FOCUS EMPLOYEE-WIN are attached to the F1 event and so on.

6.1.1 Comments

You should use comments in your dialog statements, as you do in your COBOL program, to make the code more readable. Comments allow you to:

- Clarify statements where a possibility of confusion exists with the dialog you have written.
- Disable line(s) of otherwise meaningful dialog statements.
- Separate events and procedures for readability.

Commented lines begin with an asterisk in the first column.

This sample dialog shows examples of using comments in dialog:

```
********************
 SCREENSET-INITIALIZED
*
* Save the handles of each of the bitmaps
*
     MOVE-OBJECT-HANDLE DENMARK DENMARK-HANDLE
     MOVE-OBJECT-HANDLE FRANCE FRANCE-HANDLE
*     MOVE-OBJECT-HANDLE US US-HANDLE
*     MOVE-OBJECT-HANDLE UK UK-HANDLE
     SET-FOCUS GERMAN-DLG
*********************
```

6.1.2 Levels of Dialog

You attach dialog to a window or control where an event can occur, or to the interface in general. These represent three classes of objects, but there is no essential difference between dialog applied to any of them. The dialog for each object is stored in a dialog table. The different classes of object dialog are:

- Control dialog
- Window dialog
- Global dialog

6.1.2.1 Control Dialog

This is where you attach dialog to any control except text, User Controls, ActiveX Controls and group boxes.

For example the following dialog is attached to a push button and changes the title of a window:

```
BUTTON-SELECTED
    MOVE "Customer Address Details" NEW-TITLE
    SET-OBJECT-LABEL CUSTOMER-ADDRESS-WIN NEW-TITLE
    SET-FOCUS CUSTOMER-ADDRESS-WIN
```

6.1.2.2 Window Dialog

This is where you attach dialog to any window, dialog box or tab control page.

For example the following dialog is attached to a window and handles the user selecting the Close option:

```
CLOSED-WINDOW
     SET-EXIT-FLAG
     RETC
```

6.1.2.3 Global Dialog

This is where you attach dialog to the entire screenset.

For example the following dialog initializes the first few entries of a list box:

```
SCREENSET-INITIALIZED
    INSERT-LIST-ITEM MONTH-LB "Jan" 1
    INSERT-LIST-ITEM MONTH-LB "Feb" 2
    INSERT-LIST-ITEM MONTH-LB "Mar" 3
```

6.1.2.4 Where to Locate Your Dialog Statements

The location used to define the dialog associated with a particular object or window depends on your requirements. If a procedure is:

- Called by multiple control or window objects, place it in the global dialog.
- Specific to one object, you can attach it directly to the object or to the window in which the object resides.

Well structured dialog will execute more efficiently, and is easier to maintain and debug. When you write your dialog, apply the same coding principles that apply to any programming language.

6.1.3 Types of Dialog

There are three types of dialog:

- Events
- Procedures names
- Functions

6.1.3.1 Events

For an event to be passed to a screenset, it must be defined explicitly. If an event is not defined, it will be ignored. You can define dialog events in the global dialog, in window dialog, and in individual control dialog.

Dialog System imposes a limit of 255 events for each dialog table.

An event signifies some change in the user interface. For example, it can be:

- The user pressing a key.
- A window or control receiving focus.
- A validation error occurring.
- The user moving the slider on a scroll bar.

6.1.3.1.1 How Dialog System Searches for Event Dialog

When an event occurs, Dialog System searches for the event, looking first in the control dialog, then in the dialog for the window which contains the control, and finally in the global (screenset) dialog.

Note: The same rules apply to procedures. If you call a procedure name in a control's dialog and it is not found there, the window dialog is searched for the procedure and then the global dialog.

If Dialog System does not find the event listed in any of these three places, it then looks for the event ANY-OTHER-EVENT at the control level, the window level, and then the global level.

If ANY-OTHER-EVENT is not found, no action is taken.

When Dialog System finds an event, the functions attached to it are processed step by step. Processing continues until the end of the functions listed under the event is reached, unless control branches to another procedure. Processing is delimited by another event. Dialog System processes events one at a time, queuing events and processing them as required. Dialog System imposes a limit of 255 events for each dialog table.

Dialog System events are controlled by the Panels V2 module. This module recognizes when an event occurs and processes it accordingly. Normally, the calling program does not receive information about these events from Panels V2. However, you can have these events returned to the calling program via the **ds-event-block**, found in the copyfile **dssysinf.cpy**. This file should be copied into the Working-

Storage Section of the calling program and specified in the call to Dialog System. You might want to do this where a screen event is not explicitly defined in Dialog System.

You can find a complete description of events in the topic *Dialog Statements: Events* in the Help.

Note: If a window is a secondary window, the primary window dialog is not searched.

6.1.3.2 Functions

Functions are instructions to Dialog System to do something. They operate when:

- An event they are listed under occurs.
- A procedure they are listed under is executed.

The number of functions you can enter under each event or procedure is limited by the size of the buffer for functions and events (2K), and the maximum size of the dialog table (64K).

You could enter up to 2048 functions, depending on the function, but for average length functions, a more practical limit is 341. If you need more than this, you can call a procedure containing the extra functions. For example:

```
...
    function-340
    function-341
    EXECUTE-PROCEDURE FUNCTIONS-342-TO-350
    ...
  FUNCTIONS-342-TO-350
    function-342
    function-343
    function-344
    ...
```

where `function-340` through `function-344` are some of the Dialog System functions listed in the Help.

Dialog System has a comprehensive set of functions. Some are specific to moving around screensets, for example SET-FOCUS or INVOKE-

MESSAGE-BOX. Some are similar to other programming language instructions, for example MOVE data from one data item to another.

You can find a complete description of the functions in the topic *Dialog Statements: Functions* in the Help.

6.1.3.3 Procedures

A procedure is an arbitrary name with a set of functions listed under it; in other words, a subroutine. You can think of a procedure as an event; in this case, the event "happens" when an EXECUTE-PROCEDURE or BRANCH-TO-PROCEDURE function is activated.

As an example, the following fragment of dialog shows how to code a "loop" using procedures. This loop can be used in processing lists.

```
...
BUTTON-SELECTED
    MOVE 1 INDX
    BRANCH-TO-PROCEDURE CLEAR-LOOP
  CLEAR-LOOP
    MOVE "    " SELECTED-ITEM(INDX)
    INCREMENT INDX
    IF= INDX MAX-LOOP-SIZE EXIT-CLEAR-LOOP
    BRANCH-TO-PROCEDURE CLEAR-LOOP
  EXIT-CLEAR-LOOP
    ...
```

6.2 Special Registers

Dialog System provides several registers and variables that you can use as parameters:

$REGISTER An internal register for general use. For example, you can use $REGISTER as an index into an array or to store temporary numeric values.

$NULL A parameter used to provide a default value or to space other parameters when no real parameter is needed.

$CONTROL The currently selected Control. It can be used in a general procedure where you do not know the name of the control. For example, you can use $CONTROL with the CLEAR-OBJECT function:

```
F3
 CLEAR-OBJECT $CONTROL
```

When the user presses **F3**, the current object is cleared.

$WINDOW The currently selected Window. For example, $WINDOW also can be used with the CLEAR-OBJECT function to clear the current window:

```
F2
 CLEAR-OBJECT $WINDOW
```

$EVENT-DATA A register that is written to by some events. For example, you can use $EVENT-DATA with the VAL-ERROR event. When Dialog System detects a validation error, the VAL-ERROR event is triggered and the identifier of the field in error is placed in $EVENT-DATA. This enables you then to set the focus on the field in error so the user can make corrections. The appendix *Fonts and Color* has a detailed example of using $EVENT-DATA this way.

See the topic *Dialog Statements: Events* in the Help for those events that write to $EVENT-DATA.

$INSTANCE An instance number that you can use to refer to a particular tab control page.

6.3 Important Dialog Events and Functions

Dialog System provides a rich set of events and functions that lets you control the behavior of your screenset. But how do you get started? How do you select menu choices? How do you return to your program and then what happens when you return to the screenset?

The following sections describe some basic functions with examples of dialog that may be appropriate to that function.

6.3.1 Initializing the Screenset

When you enter a screenset for the first time, you might want to set the default state of your screenset, setting such features as the state of radio buttons, the size of a data group, default values, or which menu options are enabled. The SCREENSET-INITIALIZED event lets you do this. As an example:

```
1 SCREENSET-INITIALIZED
2     SET-BUTTON-STATE OK-BUTTON 1
3     SET-BUTTON-STATE CANCEL-BUTTON 0
4     ENABLE-OBJECT SAVE-AS-PB
5     DISABLE-OBJECT SAVE-PB
6     SET-DATA-GROUP-SIZE SALES-GROUP 100
7     MOVE "*.*" SELECTION-CRITERIA
```

Line 1:

```
 SCREENSET-INITIALIZED
```

This event is triggered when a screenset is entered for the first time (or when the calling program asks Dialog System for a new screenset).

Lines 2-3:

```
     SET-BUTTON-STATE OK-BUTTON 1
     SET-BUTTON-STATE CANCEL-BUTTON 0
```

SET-BUTTON-STATE sets the state of a check box, a push button or a radio button. These two statements set the OK-BUTTON on and the CANCEL-BUTTON off.

Lines 4-5:

```
     ENABLE-OBJECT SAVE-AS-PB
     DISABLE-OBJECT SAVE-PB
```

The ENABLE-OBJECT function enables the SAVE-AS push button, the DISABLE-OBJECT disables the SAVE push button. You could use these, for example, when you have a new file and you want to force the user to enter a filename using the Save-as option.

Line 6:

```
    SET-DATA-GROUP-SIZE SALES-GROUP 100
```

The SET-DATA-GROUP-SIZE function defines the internal size of a data group. Any occurrence greater than this size is retained but is not accessible to list boxes.

Line 7:

```
     MOVE "*.*" SELECTION-CRITERIA
```

This statement sets the default value for an entry field.

6.3.2 Window Dialog

Once you have defined the visual characteristics of a window, you can use dialog to create and initialize the window.

6.3.2.1 Creating a Window

An example of dialog for creating a window is:

```
F2
   CREATE-WINDOW WINDOW1
```

where F2 is the event triggered when the user presses **F2** and WINDOW1 is the name of the window you want to create. This is the name you entered when you defined the window.

This dialog statement initializes the window, making it ready to show. Although nothing visible happens, the window is created in the background. You can then make the window visible by using the SHOW-WINDOW dialog statement.

Although window initialization is not very time-consuming, it does take a little time. To avoid the effect of this, you can create your windows before you need them, then when you are ready to make them visible, use SHOW-WINDOW to display the window.

6.3.2.2 Showing the First Window

One of the first functions you want to do when you start a screenset is to show the main window or dialog box for your application. By default, the first window or dialog box you define when you define the screenset is the first window object displayed at run time. There are two ways to specify a different window or dialog box as the first window.

You can use the definition software to define your application's main window or dialog box as the first window using **First window** on the **Screenset** menu. First window is a property of the screenset that defines the window object that is displayed at run time. If you use this method to specify the first window, you do not need to add any dialog to do this.

However, there may be cases where you explicitly want to set the focus on or show your application's main window. For example, you may not know which window is the main window at definition time and you want to select it at run time. Use the SET-FOCUS, SHOW-WINDOW or SET-FIRST-WINDOW function to do this. You need to be aware that if you use SHOW-WINDOW, you need to explicitly close the first window by entering:

```
SET-FIRST-WINDOW $NULL
```

before entering the SHOW-WINDOW function. If you use SET-FIRST-WINDOW, you do not need to do this.

6.3.2.3 Showing a Window

SHOW-WINDOW makes the window visible, but does not set the focus on the window. (An object is said to have the input focus (or focus) if any interaction with the keyboard or mouse is directed to that object.)

If the window has not already been created, SHOW-WINDOW creates the window automatically.

An example of the dialog for showing a window is:

```
 F3
    SHOW-WINDOW WINDOW2
```

where F3 is the event triggered when the user presses the **F3** key and WINDOW2 is the name of the window to be shown.

If the window has a parent window, it is also shown.

6.3.2.4 Unshowing a Window

UNSHOW-WINDOW makes the window, all child windows, and any controls invisible. The window still exists and does not need to be re-created.

You can use UNSHOW-WINDOW to clear a cluttered client area.

If you need to see the window again, SHOW-WINDOW brings the window, its child windows and all its controls back into view in exactly the same state as before it was unshown.

UNSHOW-WINDOW has two parameters. The first identifies the window to be unshown ($WINDOW means the currently selected window) and the second identifies the window that you want to receive the focus.

For example:

```
F4
   UNSHOW-WINDOW $WINDOW WINDOW2
```

unshows the window that currently has the focus ($WINDOW) and sets the focus on WINDOW2.

The chapter *Using the Screenset* discusses the advantages and disadvantages of using UNSHOW-WINDOW rather than using DELETE-WINDOW.

6.3.2.5 Changing the Default Parent Window

SET-DESKTOP-WINDOW sets the window specified by the parameter to be the parent of all objects that are normally children of the desktop.

As an example, you might have three primary windows that are all children of the desktop; WIN1 (with handle WIN1-HAND), WIN2, and WIN3. The function:

```
SET-DESKTOP-WINDOW WIN1-HAND
```

results in:

- WIN1 is still a child of the desktop.
- WIN2 and WIN3 are children of WIN1.

One use for this is the situation where selecting a function from the current screenset causes that screenset to be pushed onto the stack and a new screenset started. However, you want the first window of the new screenset to be a child of the current window in the first screenset.

You can do this by using the following functions:

```
...
MOVE-OBJECT-HANDLE WIN1 $REGISTER
SET-DESKTOP-WINDOW $REGISTER
RETC
```

in the first screenset. Then in your program, load the second screenset. WIN1 is now the parent window of all the windows in the second screenset.

See the chapter *Multiple Screensets* for a discussion of using multiple screensets.

The default parent window can be reset to the desktop by:

```
    SET-DESKTOP-WINDOW $NULL
```

Note: Windows that are already created are not changed. This function works only for new windows that are normally created as children of the desktop.

6.3.2.6 Deleting a Window

Visually, deleting a window is the same as unshowing a window. However, deletion removes the instance of the window, all child windows, and all controls that are attached to the window. Before another instance of this window can be referenced, it must be created.

An example of the dialog for deleting a window is:

```
 F6
    DELETE-WINDOW WINDOW2 WINDOW1
```

where WINDOW2 is the window to delete and WINDOW1 is the window to receive the focus.

Unless the user no longer needs that window for the session, for example a session log-on window, use UNSHOW-WINDOW rather than DELETE-WINDOW.

The chapter *Using the Screenset* discusses the advantages and disadvantages of using DELETE-WINDOW rather than UNSHOW-WINDOW.

6.3.2.7 Setting the Focus on a Window

The SET-FOCUS dialog sets the keyboard and mouse focus on the window. This means that any interaction with the keyboard or mouse is directed to this window.

If necessary, the function creates and shows the window before it sets the focus on it. An example of a dialog statement for setting focus on a window is:

```
F7
   SET-FOCUS WINDOW2
```

where WINDOW2 is the window to receive the focus.

6.3.2.8 Moving a Window

MOVE-WINDOW lets you reorganize the windows on the screen by moving a window from its current position to a new position. For example, if you create a window to use as a monitor and display a set of values, you can use MOVE-WINDOW to move the window, rather than a function that hides it, when another window is displayed.

As an example of the syntax:

```
F8
   MOVE-WINDOW $WINDOW 2 5
```

where $WINDOW indicates the current window, 2 indicates the "right" direction and 5 indicates the number of system units by which to move the window. Refer to the description of MOVE-WINDOW in the Help for a definition of these parameters.

6.3.2.9 Changing the Title of a Window

If you have multiple windows that perform similar functions, you can use a single window and change the window title using SET-OBJECT-LABEL. An example of the syntax is:

```
F9
   SET-OBJECT-LABEL WINDOW1 NEW-TITLE
```

where WINDOW1 identifies the window and NEW-TITLE is a data item that contains the new name.

Note: SET-OBJECT-LABEL does not automatically show the window if the window is in an unshow state. However, if the window is displayed, the change in title is immediately apparent.

6.3.2.10 Closing the Window

One of the options in the window system menu is **Close**. If the **Close** option is selected, Dialog System automatically closes the window. However, you can also perform additional actions if this option is selected - for example, terminate the application. The CLOSED-WINDOW event enables you to do this. As an example, the dialog:

```
 CLOSED-WINDOW
     SET-FLAG EXIT-FLAG
     RETC
```

detects the "Close" event, sets a flag indicating the user wants to terminate the user interface, and returns to the calling program. Your program could then close files and do any other necessary processes to terminate the application.

6.3.3 Pressing Buttons

BUTTON-SELECTED is the primary event associated with a push button, check box or radio button. It is triggered when you click on a button, or a check box has been selected.

For example:

```
BUTTON-SELECTED
    MOVE 0 ORD-NO($REGISTER)
    MOVE 0 ORD-DATE($REGISTER)
    MOVE 0 ORD-VAL($REGISTER)
    MOVE 0 ORD-BAL($REGISTER)
    UPDATE-LIST-ITEM ORDER-BOX $NULL $EVENT-DATA
    RETC
    REFRESH-OBJECT-TOTAL
```

When the BUTTON-SELECTED event is triggered, all the functions associated with that event are performed.

6.3.3.1 Setting and Getting Button States

The SET-BUTTON-STATE and GET-BUTTON-STATE functions enable you to control the state of buttons and check boxes. To set the state of the buttons the first time you enter the screenset, see the section *Initializing the Screenset*.

You can also set and retrieve the states of buttons any time while the screenset is running. For example, the following dialog fragment sets the state of a radio button based on some action performed in the calling program:

```
BUTTON-SELECTED
    MOVE 2 ACTION-CODE
    RETC
    SET-BUTTON-STATE FILE-ACCESSED-RB 1
    ...
```

You can also control the state of a check box by moving values of 1 or 0 to its Master Field. For example, if you have a data item named `check-credit` that is the master field tied to a check box, you can change the state of the check box by moving a 1 or a 0 to `check-credit`. Assuming the check box object is already created, you must refresh it to display its change of state.

6.3.4 Menu Bar Dialog

You can also define dialog to respond to the user selecting one of the actions of a pulldown menu. However, before you can add dialog to an action item, the action must have a name. See the topic *Objects and Properties* in the Help for details on naming actions.

As an example, the following dialog is taken from the Saledata sample application, where INSERT-CHOICE, CHANGE-CHOICE and DELETE-CHOICE are the names assigned to the menu bar actions. In the dialog, the "@" in front of the choice indicates that this is a menu choice.

```
@INSERT-CHOICE
    IF= $REGISTER 0 NO-SELECTION-PROC
    CLEAR-OBJECT INSERT-DB
    SET-FOCUS INSERT-DB
@CHANGE-CHOICE
    IF= $REGISTER 0 NO-SELECTION-PROC
    MOVE SALES-NAME($REGISTER) TMP-NAME
    MOVE SALES-REGION($REGISTER) TMP-REGION
    MOVE SALES-STATE($REGISTER) TMP-STATE
    REFRESH-OBJECT CHANGE-DB
    SET-FOCUS CHANGE-DB
@DELETE-CHOICE
    IF= $REGISTER 0 NO-SELECTION-PROC
    MOVE SALES-NAME($REGISTER) TMP-NAME
    MOVE SALES-REGION($REGISTER) TMP-REGION
    MOVE SALES-STATE($REGISTER) TMP-STATE
    REFRESH-OBJECT DELETE-DB
    SET-FOCUS DELETE-DB
```

When the user selects **Change** on the menu bar the event @CHANGE-CHOICE is triggered and all the functions under the event are performed.

6.3.4.1 Enabling and Disabling Choices

Sometimes a menu choice does not apply. For example, you might have a new data file that has been updated, but not saved. Because it is a new file, it is not yet assigned a name, and you want to force the user to select a **Save as** choice (that requests a filename) rather than a **Save** choice. One way of coding the dialog for this situation is:

```
1     ...
2     XIF= FILE-SAVED-FLAG 0 DISABLE-SAVE
3     ...
4   DISABLE-SAVE
5     DISABLE-MENU-CHOICE $WINDOW SAVE-CHOICE
6     ENABLE-MENU-CHOICE $WINDOW SAVE-AS-CHOICE
7     ...
```

Line 2:

```
XIF= FILE-SAVED-FLAG 0 DISABLE-SAVE
```

XIF is a conditional function. FILE-SAVE-FLAG is a data item in the Data Block. A value of 0 indicates a file that has not been saved. DISABLE-SAVE is the procedure to perform. Thus this statement says: If the file has not been saved, perform the DISABLE-SAVE procedure.

Line 4:

```
    DISABLE-SAVE
```

The start of the procedure to perform.

Line 5:

```
    DISABLE-MENU-CHOICE $WINDOW SAVE-CHOICE
```

This statement makes the **Save** choice unavailable.

Line 6:

```
    ENABLE-MENU-CHOICE $WINDOW SAVE-AS-CHOICE
```

This statement ensures the **Save as** choice is available.

6.3.4.2 Selecting Menu Choices

Menu bar choices are used to define what will happen when you select one of the choices. For example, the following code fragment shows one way of coding the **Cut**, **Copy** and **Paste** options of an **Edit** menu choice:

```
...
@CUT-CHOICE
    MOVE 5 ACTION-CODE
    RETC
@COPY-CHOICE
    MOVE 6 ACTION-CODE
    RETC
@PASTE-CHOICE
    MOVE 7 ACTION-CODE
    RETC
...
```

Note: The @ in column one indicates that this is a menu choice.

The section *The Menu Bar* in the chapter *Window Objects* describes the dialog for handling menu bar choices, including how to enable and disable choices.

6.3.5 Validating Input

You can validate a specific field or all fields attached to a window using the VALIDATE function. For example:

```
VALIDATE SALARY-RANGE-EF
```

validates only the SALARY-RANGE-EF entry field.

The statement:

```
VALIDATE SALARY-DETAILS-WIN
```

would validate all fields on the SALARY-DETAILS-WIN window.

If an error is detected, the VAL-ERROR event is triggered. For example, the following dialog fragment illustrates one way of coding the validation of all the fields on the current window.

```
BUTTON-SELECTED
    VALIDATE $WINDOW
    INVOKE-MESSAGE-BOX ERROR-MB "All fields OK" $REGISTER
    RETC
VAL-ERROR
    INVOKE-MESSAGE-BOX ERROR-MB ERROR-MSG-FIELD $REGISTER
    SET-FOCUS $EVENT-DATA
```

See the section *Validating Entry Fields* in the chapter *Sample Programs* for a detailed explanation of the syntax.

6.3.6 Using Procedures

Typically, you use a procedure as a common routine. For example, you usually have at least two ways to initiate a Cancel function; you can use a **Cancel** button or you can use the **Esc** key. The best way to code this is by calling a common procedure if either **Cancel** is selected or **Esc** is pressed. The following dialog fragment illustrates this method:

```
BUTTON-SELECTED
    BRANCH-TO-PROCEDURE CANCEL-PROC
ESC
    BRANCH-TO-PROCEDURE CANCEL-PROC
  CANCEL-PROC
    cancel-functions
    ...
```

Dialog System provides two methods of invoking procedures; branching to the procedure (similar to a COBOL GOTO statement), and executing the procedure (similar to a COBOL PERFORM statement). To illustrate the difference, consider the following dialog fragments.

```
F1
    BRANCH-TO-PROCEDURE CLEAR-OK
    RETC

F2
    EXECUTE-PROCEDURE CLEAR-OK
    RETC
```

In the first case (when the **F1** key is selected), control is passed to the CLEAR-OK procedure. What happens next depends on the functions contained in CLEAR-OK. The RETC function will never be executed.

When the **F2** key is selected, the functions in CLEAR-OK are executed and then control is returned to the statement following the EXECUTE-PROCEDURE statement, that is, the RETC function.

Dialog System also provides a set of branching functions. The ones you use depend on whether you want to branch to a procedure or execute the procedure.

For example, the IF= function compares two values and if they are equal, the procedure is "branched to". The XIF= function also compares two values and if they are equal, the procedure is "executed".

In both cases, if the values are not equal, the statement following the IF= or XIF= statement is executed.

6.3.7 Returning Control to the Calling Program

While your screenset is running, you may need to return to your program. For example, you may need to retrieve more data from a database, update a file, or do a complex validation. The RETC function returns control to the calling program.

For example, the following dialog fragment returns to the calling program so that the program can carry out the file deletion function.

```
BUTTON-SELECTED
    SET-FLAG DELETE-FILE-FLAG
    RETC
```

6.3.8 Regaining Control from the Calling Program

Normally, on return from your program, the statement following the RETC is executed. However, you can change this behavior if you want.

Some of the actions you may want to do when you return from the program are:

- Refresh a window. For example, your program has changed some associated data items and you want to make the changes visible to the user.
- Reset the state of your screenset. For example, your program has saved a file and now you want to reset the state of the **Save** and **Save As** buttons.
- Refresh an object. For example, you may have accessed additional data from a database that you want to display in a database.
- Check on the state of the data based on what actions your program performed. For example, you may have had to do a complex validation in your program and you want to check to see the results of the validation.

As an example of the syntax, the following dialog fragment refreshes the SALES-LIST list box and displays the window after returning from the calling program.

```
 @SORT-NAME-CHOICE
     MOVE 2 ACTION-CODE
     RETC
     REFRESH-OBJECT SALES-LIST
     SHOW-WINDOW SALES-WIN
```

6.4 Events Trapped by the Windows Operating System

Some events are trapped by the Windows Operating System before they get to Dialog System.

For example, the standard Windows system menu has a set of accelerator keys that allow you to access the functions on that menu quickly. One accelerator key is **Alt+F4**, which removes the active window and all associated windows from the screen. If you define dialog for the event caused by the **Alt+F4** key press, AF4, that event is trapped by Windows, and never reaches Dialog System.

Therefore, the following dialog defined for the key press will never be executed because Dialog System does not detect the event:

```
AF4
    SET-FOCUS NEW-EMPLOYEE-WIN
```

6.5 Sample Programs

To see some of the attributes and behavior of the window object, the sample application **Objects** is included in your demonstration directory. This application consists of a COBOL program, a screenset, and an error file. When you run the application, select the object you wish to see demonstrated on the **Object** menu.

To compile, animate and run the program, see the Help for your COBOL system.

For an example of manipulating menu bar choices dynamically at run time, refer to the **Dynmenu** demonstration in your Dialog System demonstration directory.

6.6 Sample Dialog

Of course the dialog can be more complex than that shown in this chapter. As an example, the following fragment is taken from the Dialog System definition facility. (Dialog System itself is a Dialog System application.) The following piece of dialog is attached to the mouse configuration option and lets you redefine how you want the mouse configured.

```
...
 @MOUSE-CONFIG-PD
     MOVE LEFT-BUTTON TMP-LEFT-BUTTON
     MOVE MIDDLE-BUTTON TMP-MIDDLE-BUTTON
     MOVE RIGHT-BUTTON TMP-RIGHT-BUTTON
     MOVE MOUSE-DEFAULT $REGISTER
     REFRESH-OBJECT LEFT-BTN-SB
     REFRESH-OBJECT MIDDLE-BTN-SB
     REFRESH-OBJECT RIGHT-BTN-SB
     IF= 1 MOUSE-DEFAULT DS21
     IF= 2 MOUSE-DEFAULT CUA
     IF= 3 MOUSE-DEFAULT BTN3
* Must be user defined action,
     EXECUTE-PROCEDURE ENABLE-CHOICES
     SET-BUTTON-STATE USER-DEFINED-RB 1
     SET-FOCUS USER-DEFINED-RB
* DS2.1 mouse behavior,
   DS21
     EXECUTE-PROCEDURE DISABLE-CHOICES
     SET-BUTTON-STATE DS21-RB 1
     SET-FOCUS DS21-RB
* CUA  behavior,
   CUA
     SET-BUTTON-STATE CUA89-RB 1
     EXECUTE-PROCEDURE DISABLE-CHOICES
     SET-FOCUS CUA89-RB
*
   DISABLE-CHOICES
     DISABLE-OBJECT LEFT-BTN-SB
     DISABLE-OBJECT MIDDLE-BTN-SB
     DISABLE-OBJECT RIGHT-BTN-SB
   ENABLE-CHOICES
     ENABLE-OBJECT LEFT-BTN-SB
     ENABLE-OBJECT MIDDLE-BTN-SB
     ENABLE-OBJECT RIGHT-BTN-SB
   ...
```

Notice that all this functionality is within Dialog System. Control is not returned to the COBOL program.

7 Using the Screenset

The previous chapters explained how to create a data definition and the user interface of an application. This chapter describes:

- *The call interface and how it enables your screenset to communicate with your COBOL program.*
- *Adding help to your interface.*
- *Some considerations for optimizing your applications.*

7.1 The Call Interface

This section describes:

1. Generating the COBOL copyfile from a screenset.
2. The call interface - its structure and how to use it.
3. Writing the COBOL application program.
4. Debugging and animating the screenset and your COBOL program.
5. Packaging your application.

7.1.1 Generating the Data Block Copyfile

The Data Block copyfile contains the definition of the Data Block passed from the calling program to Dialog System at run time. You must include the copyfile in your calling program. The copyfile also contains version checking information.

To generate a copyfile from your screenset:

- Select the screenset configuration options determining how that copyfile is generated.

  You can configure the definition software to write the copyfile in various ways. The sort of copyfile you want depends on whether you are writing the calling program using Micro Focus COBOL or ANSI COBOL standards. See the topic *Dialog System Configuration* in the Help for information on how to configure copyfile options.

- Select **Generate/Data block copy file** on the **File** menu in the main window of the definition software.

  A dialog box appears, enabling you to select a name and path for the copyfile. By default, Dialog System uses ***screenset-name.cpb***.

- Select a name and click **OK**.

  A message box is displayed, which shows the percentage of the copyfile that has been generated and saved. The message box disappears when the save is complete.

7.1.1.1 Generating Copyfile Options

There are some options that you should consider when generating a copyfile:

- If a data item name is greater than the 30 characters allowed, the name is truncated to 30 characters.

  Dialog System inserts a comment to this effect adjacent to the item in the copyfile, quoting the full name.

- If the **Fields prefixed by screenset Id** configuration option is set on, this causes all datanames from the Data Block to be prefixed by the screenset identifier. The data item name including this prefix must be fewer than 30 characters.

  You can change this option:

  - For a specific screenset by using **Configuration, Screenset** on the **Options** menu and setting the appropriate checkbox.

  - For all screensets by either editing the configuration file **ds.cfg**, or using **Configuration, Defaults** on the **Options** menu.

- Two data item names may be identical, as a result of being truncated. This causes a failure at compilation time.

  To correct this, change one of the data item's name in the Data Definition window and re-generate the copyfile.

Note: You can also generate the Data Block copyfile by specifying a command line similar to the following:

```
dswin/g screenset-name
```

This will generate the copyfile and exit.

For more information on generating copyfiles, see the topics *Dialog System Overview*, *The Call Interface* and *Data Definition and Validation* in the Help.

7.1.2 The Structure of the Call Interface

This section describes the most straightforward way to call Dialog System. If you are new to Dialog System, this is all you need to know to develop simple applications.

The structure of the Dialog System call interface is very simple and consists of two blocks of information that are passed between the calling program and Dialog System:

- The Control Block - corresponding to `DS-CONTROL-BLOCK` in the control block copyfile - used to carry control data between the calling program and Dialog System.
- The Data Block - corresponding to `DATA-BLOCK` in the Data Block copyfile - used to carry user data between the calling program and Dialog System.

To use your screenset, your COBOL program must make a call to the Dialog System run-time module **Dsgrun** using the Dialog System Control Block and the Data Block for your screenset. The basic call to Dialog System has the following format:

```
    CALL "DSGRUN" USING DS-CONTROL-BLOCK,
                        DATA-BLOCK
```

The run-time system searches for the screenset first in the current directory and then in the directories specified by the COBDIR environment variable.

If you generate the copyfile with **Fields prefixed by screenset ID** on in the Screenset configuration dialog box, the Data Block will be prefixed by your screenset name. Therefore you must include this prefix in the call to Dialog System. For example, the sample program **entries.cbl** in your installation **DialogSystem\demo** directory uses a screenset with the screenset identifier of "entry", and makes the call as follows:

```
CALL "DSGRUN" USING DS-CONTROL-BLOCK,
                    ENTRY-DATA-BLOCK
```

After making the initial call to Dialog System you might like to specify a routine to use if the call fails:

```
IF NOT DS-NO-ERROR
    MOVE DS-ERROR-CODE TO DISPLAY-ERROR-NO
    DISPLAY "DS ERROR NO: " DISPLAY-ERROR-NO
   PERFORM PROGRAM-TERMINATE
END-IF
```

The actual position of the call to Dialog System in your COBOL program is not critical. It is good practice to place it in a separate routine that can be called whenever your program needs to call Dialog System.

The topic *The Call Interface* in the Help describes the options you can specify in the Control Block to control how your screenset executes.

For more advanced ways to use the call interface, see the chapters *Advanced Topics* and *Multiple Screensets*.

7.1.2.1 Controlling the Use of Screensets

The way in which your calling program controls screenset handling should be considered in detail when you design your screenset and calling program. You should consider issues such as:

- How to divide functions and whether they should be grouped into separate screensets.
- Whether to provide different screensets to groups of users with different access to data, as a security consideration.

- Whether to use multiple instances of a screenset so that a user could display, compare or edit data at the same time.

For more information on the basics of screen control using the Dialog System call interface, see the topic *The Call Interface* in the Help.

7.1.2.2 Using Multiple Screensets

You can use multiple screensets by pushing and popping them from the screenset stack. By definition, this is a first in, last out operation. Pushing and popping screensets is useful to:

- Remove a screenset used for a particular function from the display when it is no longer required.
- Load multiple screensets during program initialization, and push and pop (or use) them when you need them.
- Keep the display uncluttered by windows that are not being used or do not have input focus.

There are no pre-conditions for pushing a screenset onto the screenset stack, and any screenset, or occurrence of a screenset, can be pushed or popped. Pushed screensets are normally stacked in memory, but if memory is short they will be paged to disk.

To push a screenset onto the screenset stack and start a new screenset, call Dsgrun using the following:

```
move ds-push-set to ds-control
call "dsgrun" using ds-control-block,
                    data-block
```

`ds-push-set` places the value "S" in `ds-control`. The existing screenset will be pushed onto the screenset stack. When you pop a screenset off the screenset stack you can use either of the following:

- `ds-quit-set`, which closes the existing screenset and pops the first screenset off the top of the screenset stack.
- `ds-use-set`, which pops the specified screenset off the screenset stack without closing the existing screenset.

See the chapter *Multiple Screensets* for further information.

7.1.2.3 Using Multiple Instances of the Same Screenset

Dialog System enables your calling program to use multiple instances of the same screenset. This function is useful when working with data groups where each group item has the same format. Using multiple instances of the same screenset, you can have one screenset to display, compare or update the group items as required.

Using multiple instances of the same screenset requires your program to:

- Track the number of instances of a screenset.
- Ensure that the Data Block being passed to Dsgrun is the correct one for that screenset instance.

When multiple instances are used and a screenset is first started by an `"N"` or `"S"` call, an instance value is allocated and placed in the `ds-screenset-instance` Control Block field.

The instance value is unique to that particular screenset instance. Your application must keep track of instance values because they are not assigned in any particular order.

You can track the active instance by examining `ds-event-screenset-id` and `ds-event-screenset-instance-no` within **dssysinf.cpy**, which must be copied into your program Working-Storage Section.

For more information on **dssysinf.cpy**, see the chapter *Using Panels V2*.

To specify that you want a new instance of a screenset to be loaded, set `ds-control` to `ds-use-instance-set` when you call Dsgrun.

There is no support for using multiple instances of a screenset when running through the Screenset Animator, using the definition software. If, however, Dsgrun is called from the application then multiple instances are supported.

See the chapter *Multiple Screensets* for further information.

7.1.3 Writing the COBOL Application Program

The Data Block copyfile contains not only the user data but also some version numbers that Dialog System checks against the screenset when it is called. To do this, the calling program must copy them into data items in the Control Block before it calls the Dialog System run-time.

7.1.3.1 The Control Block

The Control Block is used to carry control information and data between the calling program and Dialog System.

The Control Block consists of:

- Version checking items that check the version of the Data Block, Control Block and screenset.
- Input fields used to instruct Dialog System to perform functions.

  For example, the names and values of the constants that you use to control the screenset.
- Output fields that return values.

  For example, error codes and validation fields.

Dialog System is supplied with three versions of the Control Block copyfile:

- **ds-cntrl.ans**

  Use with ANSI-85 conformant COBOL.
- **ds-cntrl.mf**

  Use with Micro Focus conformant COBOL (COMP-5).
- **dscntrlx.mf**

  Use with Micro Focus conformant COBOL (COMP-X). Use by moving 3 to DS-VERSION-NO instead of VERSION-NO.

When you write your calling program, you must copy the copyfile into the program Working-Storage Section using the statement:

```
COPY "DS-CNTRL.MF".
```

If you are using ANSI-85 conformant COBOL, you should use the copyfile ***ds-cntrl.ans***.

The information in the data-block field, together with the Control Block, is passed backward and forward between Dialog System and the calling program whenever control is passed from one to the other.

You also need to make sure that the Control Block contains the name of the screenset and other information that controls Dialog System behavior.

Dialog System enables the user to decide the functions the program performs next, rather than the program dictating user actions. The flags set in the Data Block returned from Dialog System contain values caused by the user's action and tell the program what to do next.

The program can respond in a variety of ways including:

- Modifying stored information.
- Retrieving additional information from a database.
- Displaying results or error messages.
- Providing assistance in the use of the application.

  In a simple application, help might be handled entirely by a message box in Dialog System. An alternative way of handling help would be to write a section of code in the program to provide help whenever a Help flag was set.
- Validating the information entered by the user.
- Requesting additional input from the user.

  By returning to Dialog System after saving or clearing the record.

You can see a COBOL program **entries.cbl** that uses the sample screenset (created in the chapter *Tutorial - Creating a Sample Screenset*) in the chapter *Tutorial - Using the Sample Screenset*. This program is provided with your Dialog System software as a demonstration program.

7.1.4 Debugging and Animating the Screenset and Your COBOL Program

Net Express provides an integrated editing, debugging and animating environment.

When you are debugging an application, the source code of each program is displayed in a separate window. When you animate the code, each line of the source is highlighted in turn as each statement is executed, showing the effect of each statement. You can control the pace at which the program executes and can interrupt execution to examine and change data items. See the topic *Debugging* in the Help for more information.

After testing your program, you can change the screenset independently of the program. You can continue doing this until you have a combination of screenset and program that meets your requirements.

7.1.4.1 Testing the Screenset

One of the major benefits of Dialog System is that it is very easy to try out parts of the interface long before the screenset is completed, and before any COBOL program exists. You can prototype an incomplete screenset or a small part of a screenset to give an impression of what it is like, or to try out an idea.

Prototyping is a major aid during interface development and enables many of its tasks to be conducted rapidly. It allows you to build a working model of the system quickly so you and your users can see, on a terminal, what the system looks like when it is operating. You can then make changes quickly and inexpensively.

You can test your screenset without dialog, to make sure that the desktop layout looks good and that you have set up the fields correctly.

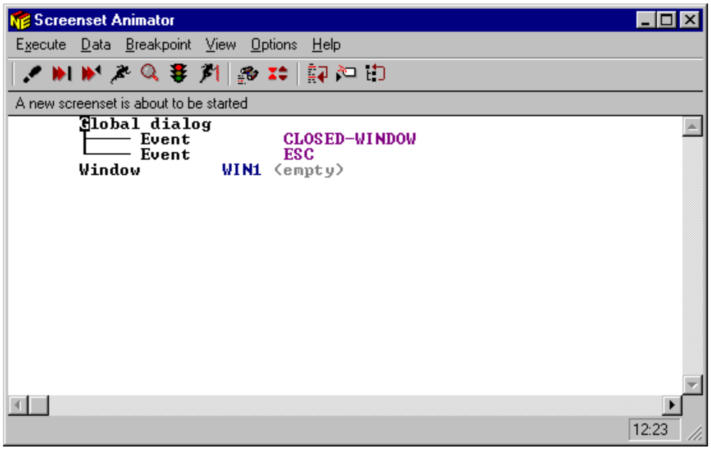
7.1.4.1.1 Using the Screenset Animator

Dialog System Screenset Animator is a utility provided with Dialog System to enable you to test and debug the screensets you create. By using the Screenset Animator, you can test the look, feel and function of a screenset even before you write the program that uses it. You can use the Screenset Animator to:

- Run your screenset independently, with fully operating objects, dialog, and a Data Block.
- Set up user data and control values, call the screenset, perform an operation, and return to inspect the returned values as many times as you want.
- Trace the execution of dialog events and functions.
- Return to definition mode and make changes to the screenset or dialog, then try the screenset again.
- Run the calling program using the Screenset Animator to check the data and control information passed between the screenset and the calling program.

Select **Debug** from the **File** menu in the main window. The Screenset Animator window is displayed as shown in Figure 7-1.

Figure 7-1. The Screenset Animator Window



A new screenset is about to be started

```
Global dialog
 ├──── Event            CLOSED-WINDOW
 └──── Event            ESC
Window            WIN1  (empty)
```

This display is explained in detail in the topic *Screenset Animator* in the Help.

To run the screenset from the Screenset Animator, select **Run** from the **Execute** menu.

When you test the screenset, you should check the main functions of the screenset with altered values, and with the default values.

When the screenset reaches a point where it should return to the calling program, it returns to the Screenset Animator where you can inspect the values it has placed in the Control and Data Blocks.

The Data Block shows the current values of the data passed into **Dsgrun**. You can alter any of the information visible at this point in the Screenset Animator to test different processing of the screenset. To alter the Data Block, you view it by selecting **Examine** from the **Data** menu, and then selecting the Data pushbutton to display your Data Block items, which can be changed via the Select pushbutton.

Again, you can change values and run the screenset again if you want, so you can simulate the action of your intended calling program.

You can change the screenset as required and re-test until you are satisfied with the interface.

See the topic *Screenset Animator* in the Help for further information.

7.1.4.1.1.1 Debugging LOST-FOCUS and GAINED-FOCUS Events

When using the screenset animator or the Net Express IDE to step through dialog functions or COBOL code executed as a result of a GAINED-FOCUS or LOST-FOCUS event, the interaction between the debugger and your application could cause additional focus changes in your application. These additional focus changes are caused by focus being set on the debugger to step through lines of dialog functions or COBOL code. While this is normal and expected behavior, you should bear it in mind when you are debugging LOST-FOCUS and GAINED-FOCUS events.

Consider, for example, an application window containing only two entry fields. With focus on the first field, the **Tab** key is pressed to move to the second field. The first field has dialog defined for the LOST-FOCUS event which causes a return to the COBOL calling program. If a COBOL breakpoint has been set on the line after the last call to DSGRUN, when the breakpoint is encountered, focus will switch from the application being debugged to the Net Express IDE. This focus change will cause an additional LOST-FOCUS event for the second entry field (which previously had focus before the COBOL breakpoint was hit). If the second field also has dialog defined for the LOST-FOCUS event, that dialog will be executed as soon as control is returned to DSGRUN. If the COBOL breakpoint had not been encountered, the additional dialog functions executed for the LOST-FOCUS on the second field would not have been executed.

7.1.4.2 Defining Dialog

A user interface is more than just a graphical display. A complete specification also describes how the user and computer interact and how the user interface software interacts with the application software.

Once you have defined the appearance of the display, you must define this run-time interaction between the user and the machine. This interaction is called dialog. Dialog consists of events and functions. When an event occurs, the functions associated with the event are executed. Events can be caused by a key on the keyboard being pressed, or a menu choice or an object being selected.

For example, a Dialog System event such as BUTTON-SELECTED occurs when a user selects a push button (with the mouse or with the keyboard). If the button selected is **Enter**, the function associated with it might be CREATE-WINDOW, to create a new window for the user to enter more information.

Dialog System lets you create or customize the dialog between the user and the display objects. Control dialog affects an individual control, window dialog affects all the controls in an individual window or dialog box, and global dialog affects all windows and objects. When an event occurs, Dialog System searches first in the relevant control dialog, then the relevant window dialog, then the global dialog.

For more information on dialog statements, see the chapter *Using Dialog* and the topic *Dialog System Overview* in the Help.

7.1.4.3 Testing the Screenset Again

Save the screenset again, and try running the interface again. Enter data into the fields and choose appropriate radio buttons, list items and other objects in your interface.

7.1.4.4 Changing the Screenset

After testing your screenset again, you may want to make changes to it (for example, to improve the screen layout). You can repeat any of the steps described in this process until you are satisfied with the screenset.

Now you need to write the COBOL program that will use the user interface contained in your screenset.

The Windows GUI Application Wizard can do this automatically if required. See the chapter *Windows GUI Application Wizard*. The chapter *Sample Programs* contains sample code to produce a very simple program that will read the user inputs and store or clear them as indicated by the user.

7.1.5 Packaging Your Application

To create the finished application you must complete various subtasks.

You use the Project facility from within Net Express to build your Dialog System application. See the topic *Building Applications* in your Help for further details.

The topic *Compiling* in the Help explains what you must do next to prepare your application for production.

When testing is completed, you are ready to assemble the finished product. A finished product can be copied onto diskettes, sent to a customer and loaded onto another machine to run as an application.

Depending on the size of the application, the finished product consists of one or more of the following:

- Executable modules. These are in the industry standard **.exe** and **.dll** file format.
- Run-time support files. These files perform functions such as file I/O and memory management.

7.2 Adding Help

You can display Windows Help directly from your Dialog System screenset using a Dialog System extension. Dialog System Extensions is the term given to dialog functions implemented by using the CALLOUT dialog function. Dialog System extensions are supplied to enable you to perform many regular programming tasks, such as displaying Windows Help or providing a file selection facility. For more information about Dialog System extensions see the topic *Dialog System Extensions* in the Help.

The Helpdemo screenset uses the Dsonline Dialog System extension, which is the Dialog System extension that displays Windows Help. The section *Adding Help* in the chapter *Tutorial - Creating a Sample Screenset* discusses the Helpdemo screenset in detail.

7.3 Optimizing the Application

You can find some hints and tips on how to optimize your COBOL program in the topic *Performance Programming* in the Help.

This section offers some additional hints for optimizing your Dialog System application. These include:

- Optimizing directory search time using the COBDIR environment variable.
- Optimizing event dialog and procedure searching.
- Using UNSHOW-WINDOW rather than DELETE-WINDOW.
- Minimizing the naming of objects.
- Using Run-time Save Format.
- Setting the `ds-no-name-info` flag.

7.3.1 Limiting the Directory Search

The Path statement tells the operating system to search specific directories on specific drives if a program or file it needs is not in the current directory. Path defines search paths in the order the alternatives are defined.

For the development environment, your Windows system registry is used by the COBOL run-time to identify the location of your Dialog System installation and your current Net Express project.

When you complete the application and it is ready for production, you can modify the Path environment variable so the directories needed by your application are nearer to the beginning of the Path alternatives.

7.3.2 Searching for Event Dialog

The chapter *Using Dialog* described the three hierarchical levels of dialog that Dialog System uses when searching for events and procedures:

- Control level.
- Window level.
- Global level.

That is, when an event occurs on a control, for example a push button, Dialog System looks to see if the event is listed in any dialog attached to the control itself. If it finds the event, it carries out the functions listed underneath.

If not, it looks for the event in the dialog attached to the window containing the control. If it does not find the event listed there, it searches in the global dialog.

You should try to:

- Keep the dialog and procedures at as low a level as possible.
- Reserve global dialog for common dialog and procedures.
- Keep the most common events near the top of the event table.

Of course, the more dialog you have, the slower the execution speed. It is also slowed down by having more events or procedures defined at the global level. Global dialog is useful for collecting common routines, but it also means that it is searched whenever an event occurs that is not found at a lower level.

7.3.3 UNSHOW-WINDOW versus DELETE-WINDOW

The UNSHOW-WINDOW and the DELETE-WINDOW functions are similar. They both make the specified window or dialog box invisible and set the input focus elsewhere.

The DELETE-WINDOW function deletes the window and its controls. If you want to show that window again, it must be created again, either explicitly or implicitly. (If a window is not created, the SHOW-WINDOW and SET-FOCUS functions do an implicit create.) Creating dialog boxes and windows (plus all the objects defined on them) does take a little time.

However, the UNSHOW-WINDOW function leaves the window created so it can be shown again quickly.

Therefore, if you have a window or dialog box that you want to make invisible but show it again later, use the UNSHOW-WINDOW function. However, if you are finished with the window or dialog box, such as a system logon window, use the DELETE-WINDOW function, which will free up some resources.

7.3.4 Minimize Naming of Objects

Some objects, such as pushbuttons or check boxes, do not need to be named unless they are referenced in dialog. By only naming those objects that are referenced in dialog, you free a small amount of storage and reduce the time Dialog System takes in searching for objects.

7.3.5 Run-time Save Format

If you distribute your Dialog System application, consider using the Run-time Save Format option. This option enables you to save enough information to run the screenset, but not enough to retrieve and edit it.

This option addresses some security issues and also reduces the amount of disk space needed for your application. The size of a reduced screenset can be as small as one third of the original screenset size.

You also might see a slight performance increase because Dialog System does not have to load so much information.

Note: Once you save your screenset with this option set, you cannot make changes to it. Therefore, is is important to keep a copy of your screenset that you can edit.

7.3.6 Using ds-no-name-info

If the `ds-no-name-info` flag in the Control Block is set, Dialog System does not read the screenset heap to get values for `ds-object-name` and `ds-window-name` on returning to your calling program. If you are **not** using either of these two fields, set this flag to true at initialization. For example:

```
move 1 to ds-no-name-info
```

This results in slightly faster exits when you return to your application from Dialog System, for example after the RETC function.

Note that setting this option will prevent you from using Screenset Animator to debug your screensets, as Screenset Animator needs the name information stored in the screenset.

7.4 Further Information

For information on more sophisticated ways of using the call interface, such as the use of multiple screensets, and multiple instances of the same screenset, see the chapter *Multiple Screensets*.

If you are migrating applications to Net Express from non Win32 environments, see the chapter *Migrating to Different Platforms* for help with cross-environment issues.

8 Windows GUI Application Wizard

The previous chapters have given you an introduction to the basic elements of Dialog System. This chapter describes how the new Windows GUI Application Wizard provides you with:

- A quick and easy method of creating a screenset with features such as a toolbar, menu bar and status bar.
- An easy route for accessing and querying any installed database.

  This is done by selecting the data access option and defining an SQL query in the Wizard.

As well as creating a new screenset, the Wizard process automatically generates associated COBOL programs configured to the functionality you have requested. Both the screenset and associated COBOL programs are automatically added to your project if required.

Note: The programs and files that are output as a result of using the Windows GUI Application Wizard are meant as a starting point for developing your own applications. They are not intended to be universally applicable to all situations, but are provided so that you can learn to use the basics. You can then adapt the code provided to suit your own needs.

The chapter *Creating a Windows GUI Application* in the *Getting Started* on-line book is a tutorial which includes practical details on how to use this wizard.

8.1 Starting the Wizard

There are three ways of starting the Windows GUI Application Wizard:

- Directly from the Net Express IDE by selecting **New** on the **File** menu and then selecting **Dialog System Screenset**.
- Directly from the IDE by selecting **New** on the **File** menu and then selecting **Project**. From the choice of project types, select **Windows GUI Project**.
- From Dialog System by selecting **New** on the **File** menu.

Note: If you open the Wizard from the IDE, and you do not have a project open, the Wizard creates a new project. You need to supply the project's name and location.

8.2 Using the Wizard

This section gives you a brief description of the features in each step of the Wizard.

8.2.1 Step 1: Screenset Name

The Wizard offers you a name for your new screenset. You can change this. You do not have to append **.gs** to the screenset name, as this will be done automatically.

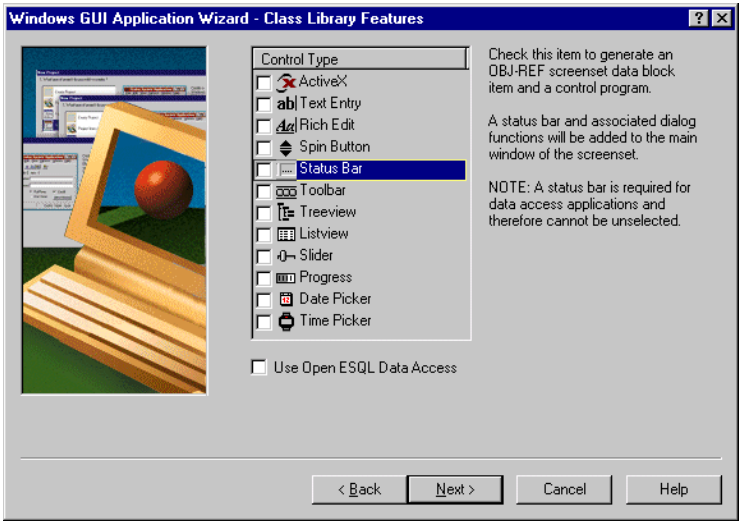
8.2.2 Step 2: Interface Type

Here you select whether your new screenset is to have a Multiple Document Interface (MDI) or a Single Document Interface (SDI). If you require an MDI application, you specify how many MDI children the interface is to have. If it is an SDI, you specify how many primary windows are required.

8.2.3 Step 3: Class Library Features

You can select features for your new screenset such as a Status Bar and a Main window toolbar. For all of the features requested, appropriate Data Block, Dialog and controlling programs can be generated. This step in the Wizard is shown in Figure 8-1.

Figure 8-1. Class Library Features



Check this item to generate an OBJ-REF screenset data block item and a control program.

A status bar and associated dialog functions will be added to the main window of the screenset.

NOTE: A status bar is required for data access applications and therefore cannot be unselected.

Selecting **Use OpenESQL Data Access** enables you to access any installed ODBC datasource and to set up queries as described in the next step. If you leave this option unchecked, you skip the next step and go directly to Step 5.

When you select **Use OpenESQL Data Access**, a status bar is automatically selected for your screenset.

8.2.4 Step 4: Defining a Query

If you have not selected **Use OpenESQL Data Access** in Step 3, you skip this step and go directly to Step 5.

Note: If you did not install the ODBC Drivers at installation time, you need to do so before you can use this part of the Wizard.

In this step you can:

- View, in the left-hand pane, a list of all the ODBC data sources that you have registered on your system.

  Access to installed data sources uses the OpenESQL Assistant technology.

  If you haven't set up any data sources yourself, you can use one of the sample data sources that is set up automatically when you install Net Express. One of these, **NetExpress Sample2** points to a sample Microsoft Access database, **sample.mdb** which is supplied with Net Express and is installed in the **base\demo\smpldata\access** directory.

- Select a table.

  If you double-click on a database name, you see a list of tables contained in the database. Select a table from this list by double-clicking on the table.

- Display columns.

  When you select a table, the table expands into columns. Double-click on a column to select it. When selected, an *EXEC SQL* statement is added into the right-hand **Query** pane.

- Select multiple tables.

  If you double-click on a subsequent table, the Table Added To Query dialog box tells you how the tables will be joined. If you want to use a different join, click **No** and use the **Search Criteria** tab.

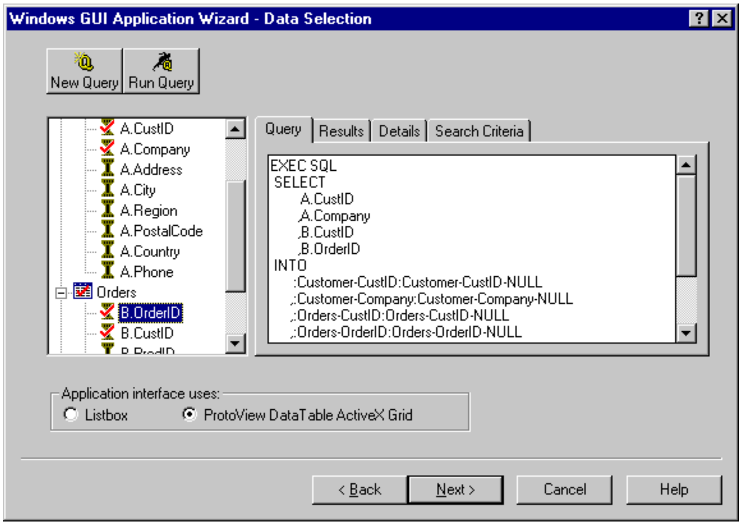
- Select a query.

  Once you have selected the columns, you can run the query by clicking **RunQuery** on the toolbar and view the query results to verify the data that will be accessed by your generated application.

Note: You must select at least one primary key. If you do not select a primary key, one is automatically selected for you.

The screen for this step in the Wizard is shown in Figure 8-2.

Figure 8-2. Defining a Query



```
EXEC SQL
 SELECT
        A.CustID
       ,A.Company
       ,B.CustID
       ,B.OrderID
 INTO
      :Customer-CustID:Customer-CustID-NULL
     ,:Customer-Company:Customer-Company-NULL
     ,:Orders-CustID:Orders-CustID-NULL
     ,:Orders-OrderID:Orders-OrderID-NULL
```

Application interface uses:
○ Listbox ● ProtoView DataTable ActiveX Grid

This screen is used to build and test queries for use in your generated application. You can activate it independently from the **Tools, OpenESQL Assistant** menu in the IDE.

For further details about database access and how to use SQL, see the chapter *OpenESQL* in the *Database Access* on-line book.

8.2.5 Step 5: Extensions

In this step you set up parameter blocks in your Data Block to enable the use of any of the Dialog System extensions that you may need for your screenset. Each extension has different requirements.

8.2.6 Step 6: Dialog System Run-time Configuration Options

Selecting any of these options causes the relevant dialog code to be inserted into your generated screenset.

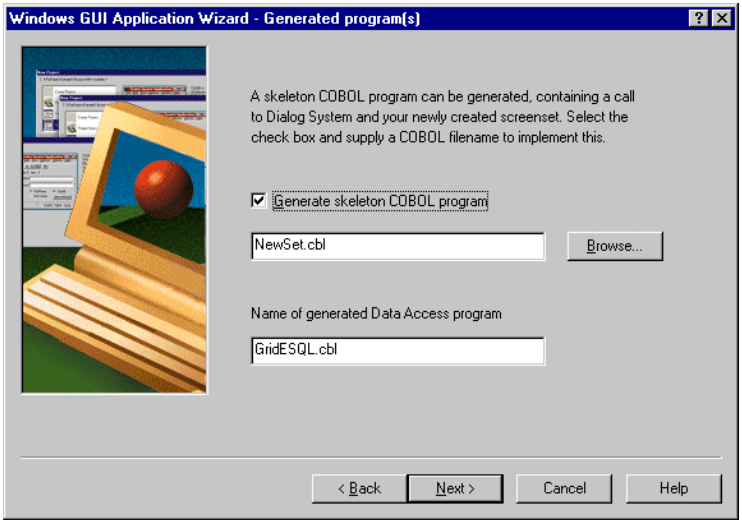
Click **Help** for full details on each option.

8.2.7 Step 7: Generate COBOL Programs

If you select **Generate skeleton COBOL program**, Dialog System will generate a skeleton COBOL program, which starts the application with a call to **dsgrun**. A default name is provided but you can enter a new name for this program.

The screen for this step in the Wizard is shown in Figure 8-3.

Figure 8-3. Generating Programs



Windows GUI Application Wizard - Generated program(s)

A skeleton COBOL program can be generated, containing a call to Dialog System and your newly created screenset. Select the check box and supply a COBOL filename to implement this.

☑ Generate skeleton COBOL program

NewSet.cbl

Name of generated Data Access program

GridESQL.cbl

You will see an entry for **Name of generated Data Access program** only if you selected **Use OpenESQL Data Access** in Step 3.

8.2.8 Step 8: Validation of Selected Options

In this step you validate the options that you have selected in the previous steps. You can return to earlier steps to amend any selections. After accepting your selections, the screenset and programs are created.

8.3 Output from the Wizard

The files which are output from the Windows GUI Application Wizard depend on the options you selected, but typically will be one or more of the following:

- An application COBOL program, with the extension **.cbl**.
- A screenset with the extension **.gs**.
- A status bar with the filename **sbar.cbl**.
- **GridESQL.cbl**, which creates and manages the grid control at run time if you selected **Grid View** in Step 4.
- **LBoxESQL.cbl**, which creates and manages the listbox control at run time if you selected **Listbox View** in Step 4.
- A toolbar control program with the extension **.cbl**.

8.4 Running the Application

To use the output from the Windows GUI Application Wizard to view, query or change the database details:

1. Rebuild the project by selecting **Rebuild All** on the IDE **Project** menu.
2. Run the application using the IDE by clicking **Run** on the toolbar or selecting **Run** on the **Animate** menu.

8.5 Further Information

Now that you have generated a data access application, you can manipulate the data. See the following chapter, *Data Access* for details.

Part 2: Advanced Features

This part contains the following chapters:

9 Data Access

The previous chapter showed you how the new Windows GUI Application Wizard can provide you with an easy route for accessing and querying any installed database. This chapter describes how you can manipulate this data.

Note: The programs and files that are output as a result of using the Windows GUI Applications Wizard are meant as a starting point for developing your own data access applications. They are not intended to be universally applicable to all situations, but are provided so that you can learn to use the basics. You can then adapt the code provided to suit your own, possibly more complex, needs.

The chapter *Creating a Windows GUI Application* in the *Getting Started* on-line book is a tutorial which includes practical details on how to use the wizard.

9.1 The Windows GUI Application Wizard

The Wizard process created a new screenset and automatically generated an associated COBOL program configured to the functionality you requested. Both the screenset and associated COBOL program were automatically added to your project.

If you want to access an installed database, you must have selected the **OpenESQL Data Access** option in Step 3: Class Library Features.

9.2 Accessing Installed Databases

There are two possible ways of viewing the results of your query after building your application, depending on the option you chose in Step 4 of the Windows GUI Application Wizard:

- List box view.

  - or -

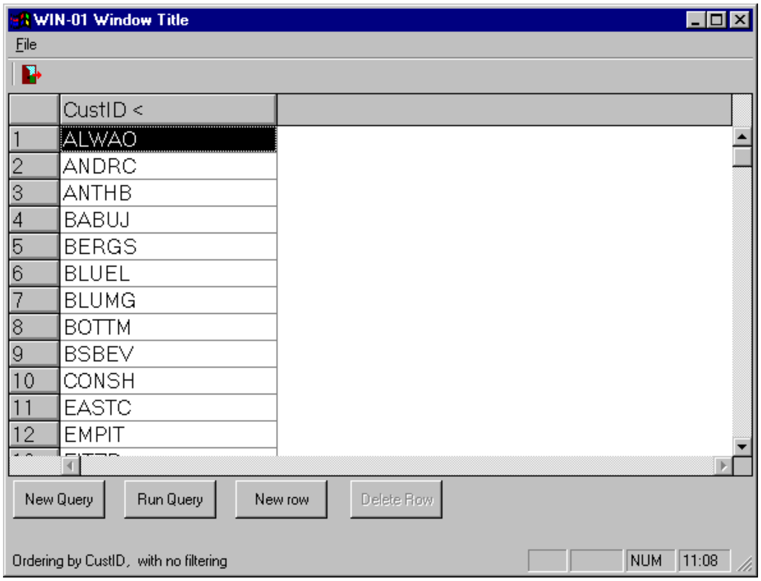
- Grid view.

These views provide the same functionality but use different interfaces. The grid view has a more direct feel in use; the list box view is more familiar to experienced Windows users.

When you run a query, all of the data for that query is read from the database into memory. Only the page of rows visible on screen is loaded from memory at any one time: if you scroll, the current page is updated by retrieving rows from memory.

The grid view of your single table database query might look like that in Figure 9-1.

Figure 9-1. Grid View of Single Table Database Query

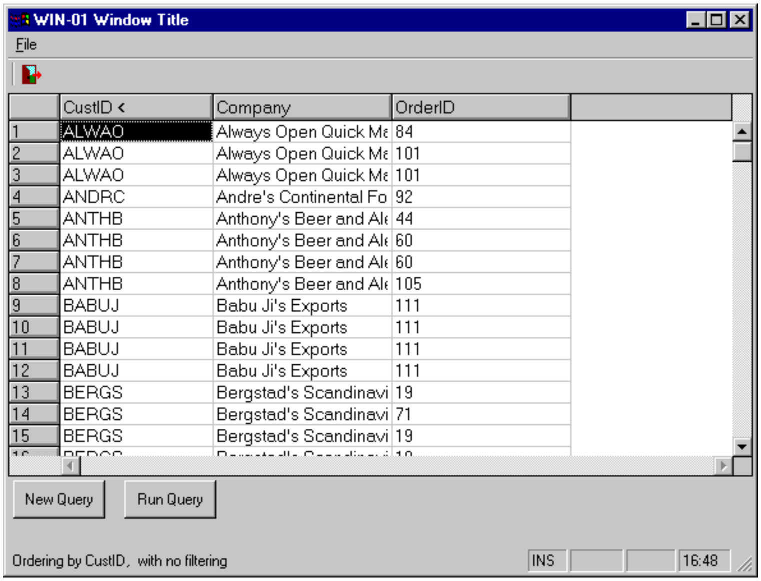


	CustID <
1	ALWAO
2	ANDRC
3	ANTHB
4	BABUJ
5	BERGS
6	BLUEL
7	BLUMG
8	BOTTM
9	BSBEV
10	CONSH
11	EASTC
12	EMPIT

Ordering by CustID, with no filtering

The grid view of your table join database query might look like that in Figure 9-2.

Figure 9-2. Grid View of Table Join Database Query

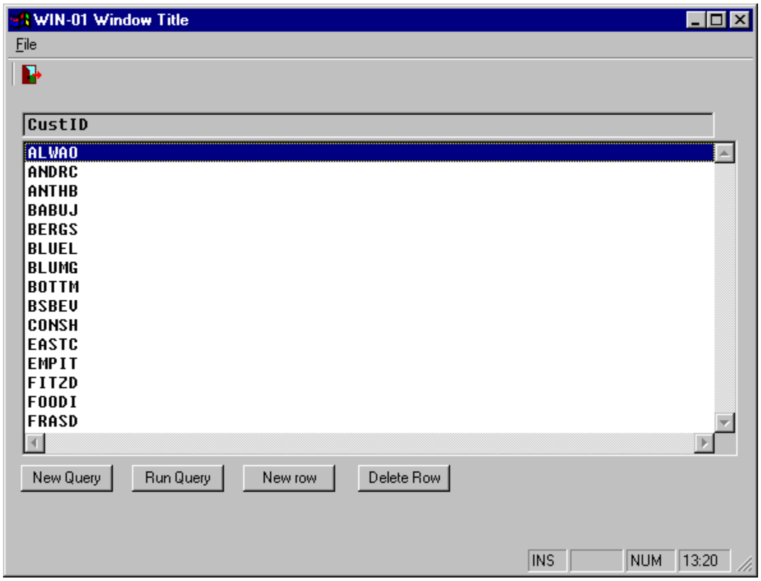


	CustID <	Company	OrderID
1	ALWAO	Always Open Quick Ma	84
2	ALWAO	Always Open Quick Ma	101
3	ALWAO	Always Open Quick Ma	101
4	ANDRC	Andre's Continental Fo	92
5	ANTHB	Anthony's Beer and Ale	44
6	ANTHB	Anthony's Beer and Ale	60
7	ANTHB	Anthony's Beer and Ale	60
8	ANTHB	Anthony's Beer and Ale	105
9	BABUJ	Babu Ji's Exports	111
10	BABUJ	Babu Ji's Exports	111
11	BABUJ	Babu Ji's Exports	111
12	BABUJ	Babu Ji's Exports	111
13	BERGS	Bergstad's Scandinavi	19
14	BERGS	Bergstad's Scandinavi	71
15	BERGS	Bergstad's Scandinavi	19
16	BERGS	Bergstad's Scandinavi	19

Ordering by CustID, with no filtering

The list box view of your single table database query might look like that in Figure 9-3.

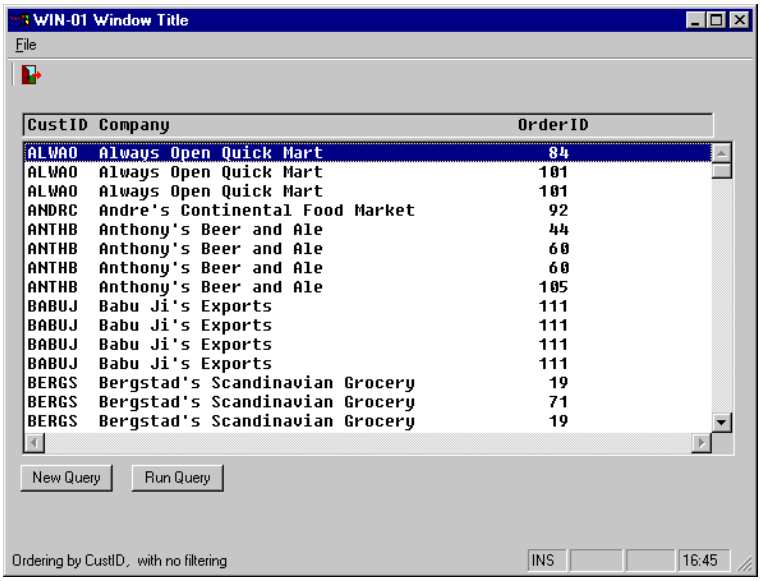
Figure 9-3. List Box View of Single Table Database Query



```
CustID
ALWAO
ANDRC
ANTHB
BABUJ
BERGS
BLUEL
BLUMG
BOTTM
BSBEV
CONSH
EASTC
EMPIT
FITZD
FOODI
FRASD
```

The list box view of your table join database query might look like that in Figure 9-4.

Figure 9-4. List Box View of Table Join Database Query



CustID	Company	OrderID
ALWAO	Always Open Quick Mart	84
ALWAO	Always Open Quick Mart	101
ALWAO	Always Open Quick Mart	101
ANDRC	Andre's Continental Food Market	92
ANTHB	Anthony's Beer and Ale	44
ANTHB	Anthony's Beer and Ale	60
ANTHB	Anthony's Beer and Ale	60
ANTHB	Anthony's Beer and Ale	105
BABUJ	Babu Ji's Exports	111
BABUJ	Babu Ji's Exports	111
BABUJ	Babu Ji's Exports	111
BABUJ	Babu Ji's Exports	111
BERGS	Bergstad's Scandinavian Grocery	19
BERGS	Bergstad's Scandinavian Grocery	71
BERGS	Bergstad's Scandinavian Grocery	19

Ordering by CustID, with no filtering

The status bar is at the foot of the screen and gives information about the object that the mouse is over. When the mouse is over the general part of the screen, the status bar text tells you which type of function you have selected. Message boxes give you information about errors, for example, trying to edit a key field.

9.3 Manipulating the Data

You can see that the table join query views have only two buttons - **New Query** and **Run Query**. This is because table join output is read-only, so the following data manipulation information applies only to single table database queries.

You can perform the following functions:

- Edit data in the database.
- Insert a new row.
- Delete a row.

9.3.1 Edit Data

	Grid View	**List View**
Change data in a field	Select the field and specify the necessary changes, remembering that you cannot change key fields. If the edited field is in a column which has been sorted, your change may result in the whole row being moved to another part of the display.	Select the row to change and specify the necessary changes in the dialog box which opens. If the edited field is in a column which has been sorted, your change may result in the whole row being moved to another part of the display.
Move to the next column	Use either the mouse or the left and right arrow keys.	
Delete data in a field	Select the field and press the **Delete** key.	Select the row to change and specify the necessary changes in the dialog box which opens. If the edited field is in a column which has been sorted, your change may result in the whole row being moved to another part of the display.
Refresh the screen	Click **Run Query**.	Click **Run Query**.

Note: Because concurrent versions of the same query can be open, the application refreshes your query from disk before echoing any edits that you make to the data. This ensures that you see the most up-to-date version of the data.

9.3.2 Insert a New Row

	Grid View	**List View**
Insert a new row	Click **New Row**. Enter the new data in the top row of the empty grid.	When you click **New Row,** you see a dialog box with entry fields for each column except for key fields. There is no validation for any of the entry fields.
Perform the insert	Press **F7**. Status Bar reports the result of your update.	Press **F7**. The Status Bar reports the result of your update.
View the new line in your query	Click **Run Query**.	Press **Enter**.

Note: Your new line may not be immediately visible. This could be a result of either of the following:

- This data may be excluded by the current query. To see the new line you need to select a new query.
- The new line is outside of the current view and you need to scroll to the new row's position.

9.3.3 Delete a Row

For both grid and list box views, to delete a row of data, select the row to be deleted and click **Delete Row**.

9.4 Viewing the Data

You can perform the following functions on all views:

- Search for data by defining a new query.
- Sort the data by column.

9.4.1 Search for Data

	Grid View	List View
Search for data	**1** Click **New Query**. **2** Highlight the field to search on. **3** Enter your search criteria. **4** Right-click in the field to see a pop-up menu. **5** Select the logical operator (=, >, < and !=). The default is 'no filter'. **6** Click **Run Query** to see the results of your search.	**1** Click **New Query**. A dialog box is shown with entry fields for each field in your application. **2** Enter your search criteria. **3** Right-click in the field to see a pop-up menu. **4** Select the logical operator (=, >, < and !=). The default is 'no filter'. **5** Press **Enter** to see the results of your search.

9.4.2 Sort Data

	Grid View	List View
Sort data in a column	Click on the column heading bar. You see: • A < symbol on the heading bar to indicate that the data is sorted in ascending order. • A > symbol on the heading bar to indicate that the data is sorted in descending order.	Right-click in the list box. You see a pop-up menu from which you select the column to sort.

10 Programming Your Own Controls

In this chapter you see how to tailor User and ActiveX Controls to use with your Dialog System application. The topics covered are:

- Control programs.

  Throughout this chapter, we use the term *control program*. This is a COBOL source program that creates and manipulates the user interface object or control.

- ActiveX Controls.
- User Controls.

10.1 Control Programs

Using a control program, any object can be manipulated using programmed functions appropriate to that object. You can perform actions on that control such as refreshing, deleting or updating data associated with the control:

- By setting the value of CALL-FUNCTION.
- By setting other parameters in the FUNCTION-DATA Data Block group.
- Using CALLOUT to your control program.

Dialog System provides programs that enable you to include the following controls in your application:

- GUI Class Library Win32 Controls.
- ActiveX Controls.

The source code for each control is designed to be as generic as possible. You can tailor and re-use the code with the minimum of changes. The extent to which this is possible, however, varies according to the control itself.

- The Status Bar and Spinbutton controls can be fully used without making any changes.
- The Toolbar and Tree View controls can be used unchanged, but you need to change the data interface to the program to suit your own requirements.
- ActiveX Controls are implementation-specific.

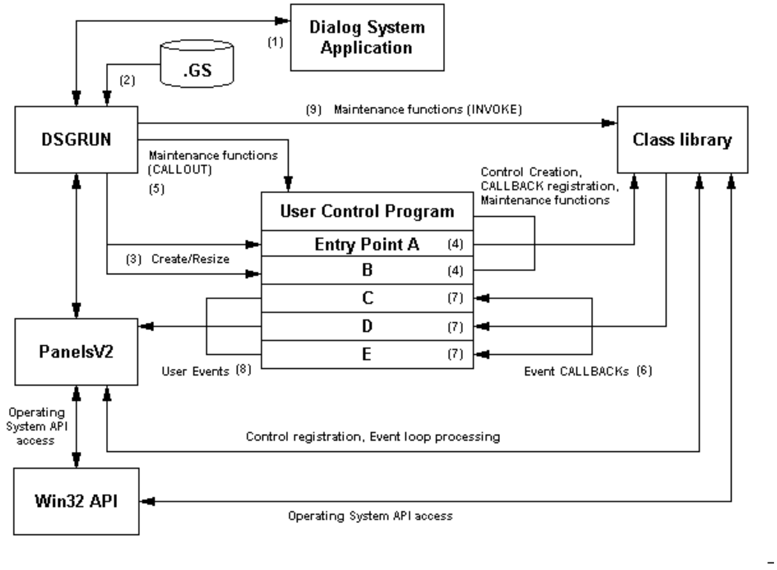
  You need to tailor these versions to meet your own requirements. Each of these controls, however, has generic code that does not need changing and you will find references to modifying the programs documented in the section *User Controls*.

The control programs are implemented using calls to the Net Express class libraries.

10.1.1 Control Implementation Architecture

The following diagram illustrates the architecture used to implement controls using Dialog System and the class libraries.

Figure 10-1. Controls Implementation Architecture


Dialog System Application
(1)
.GS
(2)
DSGRUN
(9) Maintenance functions (INVOKE)
Class library
Maintenance functions (CALLOUT)
(5)
Control Creation, CALLBACK registration, Maintenance functions
User Control Program
Entry Point A (4)
B (4)
C (7)
D (7)
E (7)
(3) Create/Resize
PanelsV2
User Events (8)
Event CALLBACKs (6)
Operating System API access
Control registration, Event loop processing
Win32 API
Operating System API access

When you create an ActiveX or user control:

1 You paint a control on a window in your screenset and associate a master field and program name with that control.

2 An application program calls **dsgrun** and supplies a screenset in the normal manner.

3 When Dialog System creates a window on which an ActiveX or user control is defined, an entry point in the program associated with the user control is called.

4 The entry point code performs all the tasks necessary to enable the class library to create the window object.

As part of the control creation process, code exists in the generated control programs to enable the registration of callbacks for events in which the program is interested. This associates program entry points with the occurrence of defined system events.

5 When the defined system event occurs, it causes the specified entry point code to be executed.

6. The entry point code performs all the required processing, including updating the Data Block.
7. The entry point code passes a message back to the Panels V2 program. This is received as a USER-EVENT in the Dialog System screenset, so that any additional dialog processing can be performed.
8. The control can be manipulated through a CALLOUT to the user control program or directly by the class library through the Dialog System INVOKE function.

10.2 ActiveX Controls

As you saw in the chapter *Control Objects*, ActiveX Controls are supplied by third-party vendors and can be integrated into Dialog System. When you use an ActiveX Control in your interface, you need to customize it to suit your own requirements.

10.2.1 ActiveX Control Properties

There are three types of ActiveX design-time properties:

- Dialog System properties

  These are the properties which must be defined when you select an ActiveX Control.

  Select **Properties** on the **Edit** menu, or double-click on the control.

- General property list

  Select the ActiveX Control in your screenset to display this automatically.

- ActiveX property pages

  Right-click on the ActiveX Control. This includes the general properties in the general property list above. Not all ActiveX Controls have property pages implemented.

In many cases, the default properties will not suit your application. For example, the default properties for a spreadsheet ActiveX Control may only have two columns and two rows. On the other hand, some ActiveX Controls, such as the Dialog System clock ActiveX Control, has mainly acceptable defaults.

You should refer to the documentation for the ActiveX Control to determine the properties that you need to set. The changed property data is stored in your screenset.

10.2.2 Tailoring Your ActiveX Control

The steps that you need to take to tailor your ActiveX Control are:

- Select which ActiveX Control you require.
- Define its Dialog System properties.
- Customize the ActiveX Control program using the Programming Assistant.

10.2.2.1 Selecting an ActiveX Control

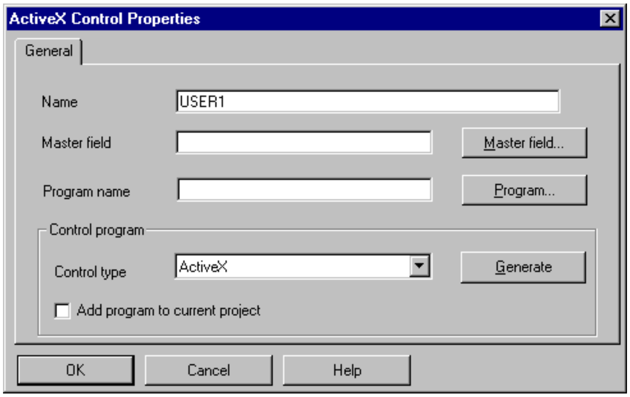
To specify the ActiveX Control directly from Dialog System's menu:

1. Select **Import** on the **File** menu.
2. Select **ActiveX Control** on the sub-pulldown menu.
3. Select your required ActiveX Control from the list box showing the ActiveX Controls which are registered on your system.

   An icon representing this control appears on the ActiveX toolbar. You can select this ActiveX Control directly from the ActiveX toolbar whenever you need to re-use it.
4. Ensure that your Data Block has an item defined as OBJ-REF with which the ActiveX Control is to be associated.
5. Position and size the ActiveX Control.

   The ActiveX Control Properties dialog box is automatically shown as in Figure 10-2.

Figure 10-2. ActiveX Control Properties Dialog Box



Name: USER1

Master field:

Program name:

Control type: ActiveX

☐ Add program to current project

10.2.2.2 Defining the ActiveX Control Properties

In the ActiveX Control Properties dialog box:

1 Specify the Data Block Master Field OBJ-REF name for this control.

2 Specify the name for the control program.

3 Ensure your Net Express project is open and available in the Net Express IDE.

4 Select **Add program to current project**.

5 Click **Generate** to generate a COBOL source program which is automatically tailored to the control you are using and added to the open project.

This will be the controlling program for the ActiveX control in your screenset.

6 Click **OK**.

10.2.2.3 Customizing the ActiveX Control Program with the Programming Assistant

An example of a customized ActiveX Control program is located in the folder **Net Express\DialogSystem\Demo\Activex\Custgrid**. The file **custgrid.txt** describes how to run the demonstration.

Customizing a control program to work with your ActiveX Control involves:

- Registering event handlers for the events that the ActiveX Control triggers.
- Implementing code to handle the required events.
- Adding code to set run-time properties and invoke methods in the ActiveX Control.

To tailor your ActiveX Control you need to explore the methods, properties and events that your ActiveX Control supports. The Programming Assistant simplifies this task by providing a visual environment from which you can extract the pre-defined functions that your program requires. It provides you with code to:

- Get or Set properties.

  An ActiveX Control has properties or attributes, for example, the background color, which you can read or change:

  - Use a GET `property name` statement to find out the current state of the property.
  - Use a SET `property name` statement to change a property.

- Invoke methods.

  Methods are a set of pre-defined functions provided by the ActiveX control. These pre-defined functions instruct the control to perform some action, during the course of which they can either send parameters or return values or do both.

- Respond to events.

  Events are provided by the ActiveX control to indicate that an action has occurred. The event may send parameter information which describes the details of the event. This information is received and recognized by the application program which is using the ActiveX control.

Registering a callback causes a block of your COBOL code to be executed when an event occurs on your control. For example, when an ActiveX grid row is to be deleted, call the program entry point detailed in `MessageName`. The entry point code exists in the controlling program source file.

For example, the following code registers a callback to be executed when an ActiveX grid data row is to be deleted by a user's interaction.

The name of the event generated by the ActiveX:

```
MOVE z"RowDeleted " TO MessageName
```

The entry point name to be executed on occurrence of the event:

```
MOVE ProgramID & z"DeletedRow" TO CodeName
```

The Callback registration:

```
INVOKE anActiveX "SetNamedEventToEntry" USING
MessageName

CodeName
```

10.2.2.4 Starting the Programming Assistant

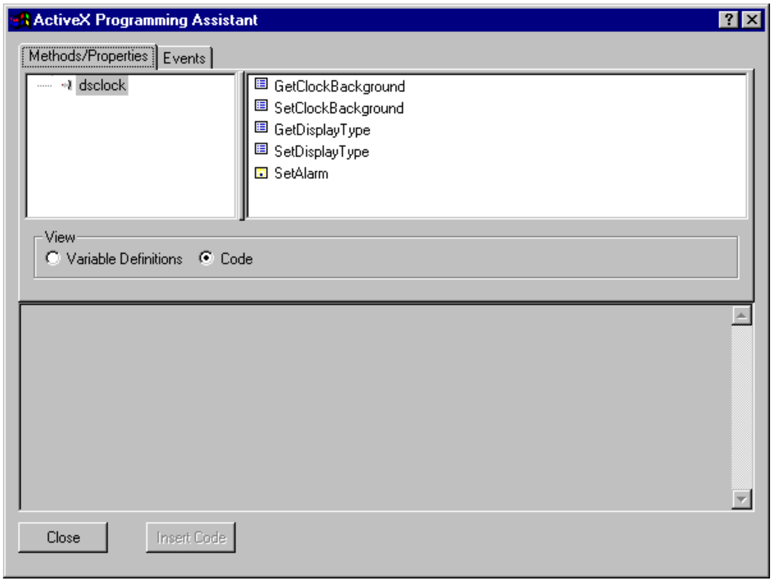
To access the Programming Assistant:

1 Right-click in the ActiveX control.

2 Select **Programming Assistant...** on the context menu.

The ActiveX Programming Assistant dialog box is shown in Figure 10-3.

Figure 10-3. ActiveX Programming Assistant Dialog Box



There are two tab pages available in this dialog box, enabling you to insert the code relating to methods and properties or events directly into your open COBOL program.

10.2.2.4.1 Methods and Properties

Selecting the **Methods/Properties** tab displays the screen shown above in Figure 10-3.

The left-hand pane displays the ActiveX control and exposes its run-time object hierarchy. The right-hand pane displays the methods and properties that this control provides.

When you select a Method or Property in the right-hand pane, the associated COBOL code appears in the pane in the lower part of the dialog box.

To insert this code into the program which creates the ActiveX at run time:

1. Open the program in the Net Express IDE.
2. Place the cursor at the required position in this program for the new code.

   There is no clearly commented place to insert this code, as methods and properties insertions depend entirely on your program logic. The code can be used in the context of control initialization, event callback or simple run-time manipulation.
3. Return to the Programming Assistant window.
4. Click on **Insert Code**.

   The created code contains all required parameters and return values needed for the function you selected.

10.2.2.4.2 Sub-objects

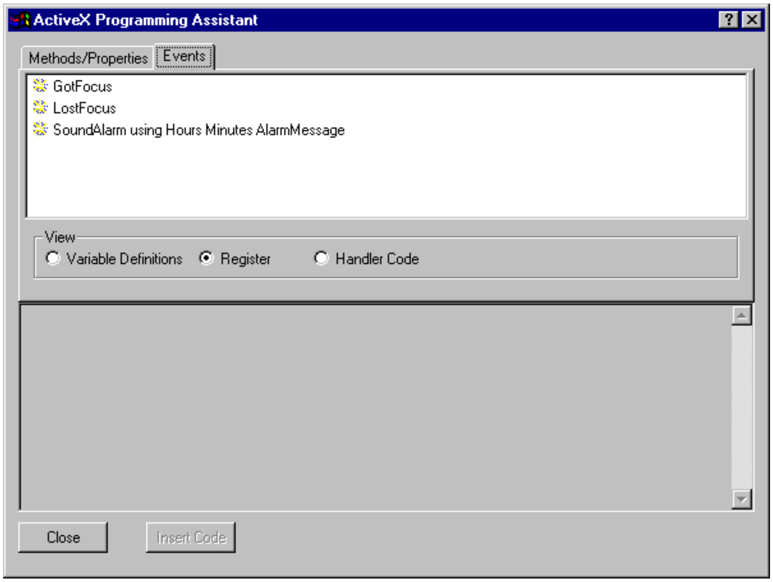
ActiveX Controls can have associated sub-objects, each of which has its own methods and properties which can be viewed or edited. To see these, click on the + sign connected with the ActiveX Control in the left-hand pane.

Sub-objects generally do not have related events. These are globally handled in the ActiveX Control.

10.2.2.4.3 Events

Selecting the **Events** tab displays the screen in Figure 10-4.

Figure 10-4. ActiveX Programming Assistant Dialog Box - Events



GotFocus
LostFocus
SoundAlarm using Hours Minutes AlarmMessage

View
○ Variable Definitions ● Register ○ Handler Code

The upper part of the dialog box shows a list of event names. Next to each event name there may be parameters which are passed by the event to the ActiveX controlling program.

As with the methods and properties view, the lower pane displays the code associated with the event selected in the upper pane. The radio buttons' functions are described below.

10.2.2.4.4 Variables

When the **Variable Definitions** button is selected, you can see the variables connected with the event selected in the upper pane. These variables:

- Match the parameter types which are passed by the ActiveX Control.

Parameters must be retrieved into the defined variables in the event handler code before being accessible to the controlling program.

- Must be inserted in the Local-Storage Section of your application program.

10.2.2.4.5 Event Registration

Selecting the **Register** button displays the code which your controlling program needs in order to register a callback for the selected event. Registering an event callback performs two tasks:

- Specifying the entry point where the event handler code is located.
- Notifying the OLE class library that it needs to recognize the event.

The generated name of the entry point is **Program ID+On+event name**. For example, if the event name is **BeforeDeleteRow** the generated entry point name would be **Program IDOnBeforeDeleteRow**.

Note: The generated entry point name is prefixed by the program ID to ensure the uniqueness of the entry point across different control programs. For example, you may have the same event name in different ActiveX controls. Without the program ID attached to the entry point, the OLE class library would not know which control program to call.

You need to insert this code in the Register-Callbacks section of your controlling program which is executed on creation of the control.

10.2.2.4.6 Event Handler

When you select the **Handler Code** button:

- The event handling code is displayed in the lower pane of the dialog box.
- The **Include Comments** checkbox is selected.

  When this is selected, the code displayed for the event includes:

  - Comments for general program instructions.
  - Commented out instructions for each parameter that can be modified, showing how to modify that parameter.

You insert the event handling code at the comment

*Add your event handler code here.

Sample code is available in the generated control program.

To insert this code:

1. Open the program in the Net Express IDE.
2. Click on the insertion position in this program for the new code.

   There is a clearly commented place to insert this code. The code can be used in the context of control initialization, event callback or simple run-time manipulation.
3. Return to the Programming Assistant window.
4. Click on **Insert Code**.

   The created code contains all required parameters and return values needed for the function you selected.

10.2.2.5 Summary

The Programming Assistant for ActiveX controls significantly reduces the ActiveX control familiarisation process, as well as the time taken to produce the code you need.

10.3 User Controls

When you use a User Control in your interface, you need to customize it to suit your own requirements. The steps that you need to take to tailor your User Control are:

- Create the User Control and define its Dialog System properties.
- Generate the controlling program appropriate to the control type you require.
- Customize the controlling program.

10.3.1 Specify the User Control

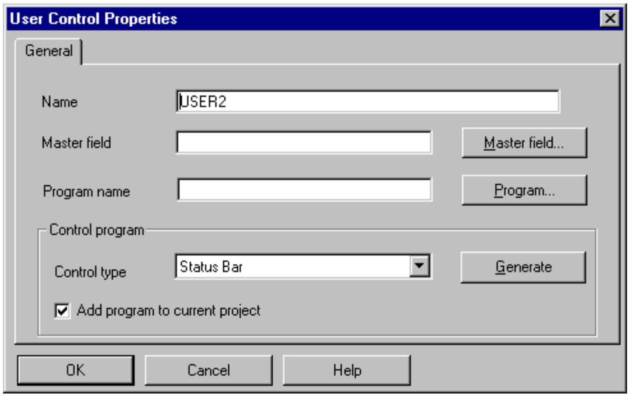
The User Control object allows you to add a Net Express Class Library GUI object to your screenset.

To create the User Control:

1. Define a Data Block entry of type OBJ-REF with which the control will be associated.
2. Select the window where you want to add the User Control.
3. Select **User control** on the **Object** menu, or click User Control on the **Objects** toolbar.
4. Position and size the user control.

   The User Control dialog box is shown as in Figure 10-5.

Figure 10-5. User Control Dialog System Properties Dialog Box



User Control Properties

General

Name: USER2

Master field:

Program name:

Control program

Control type: Status Bar

☑ Add program to current project

5. Complete the following items on the User Control Properties dialog box:

   a. Choose an appropriate name for the user control. Make this as descriptive as possible, for example MAIN-WINDOW-STATUS-BAR.

   b. Specify the relevant OBJ-REF data item as the master field name.

   c. Specify a name for your user control program.

You should choose a name which is different from any of the Class Programs defined in the Net Express class library. Once you have chosen a new name for the tailored control program, you can start to modify it to provide the functionality that you require.

d Select the type of control you require from those available in the drop down list.

e Ensure your Net Express project is open and available in the Net Express IDE.

f Select **Add program to current project**.

g Click **Generate** to generate a COBOL source program which is automatically tailored to the control you are using and added to the open project.

This will be the controlling program for the user control in your screenset.

h Click **OK**.

10.3.2 User Control Types

In all circumstances you will need to import the Data Block definitions listed in the **funcdata.imp** file found in your **DialogSystem\Source** directory. Select the Dialog System **File/Import/Screenset** menu choice and select **funcdata.imp** for import on the resulting dialog box.

The following sections detail the changes you need to make to successfully use the generated programs with your screenset.

10.3.2.1 Spin Button

No COBOL code changes are required. Simply implement a USER-EVENT dialog event on the parent window of your spin button, to respond to the event posted by the generated program. Your dialog code can then ensure your Data Block master field is updated to the new value passed by the control program.

Example:

```
USER-EVENT
     XIF=$EVENT-DATA 34580 UPDATE-MASTER
     UPDATE-MASTER
     * The User control program updates the NUMERIC-VALUE
     * field. This code updates the field I want
     MOVE NUMERIC-VALUE(1) MY-NUMERIC-FIELD
```

10.3.2.2 Status Bar

No COBOL code changes are required. You simply need to implement the following points, which are covered in more detail in the Tutorial chapters later in this book.

- Dialog code to set up MOUSE-OVER events for the hint text status bar section, as detailed in the Help.
- Dialog events to respond to window-moved and -sized events which CALLOUT to the control program to resize the status bar accordingly.
- A TIMEOUT procedure to enable a regular CALLOUT to refresh the toggle key (Insert/Overstrike, CAPS and Num Lock) states, and the current system time.

This procedure is covered in detail in the chapter *Tutorial - Adding and Customizing a Status Bar.*

10.3.2.3 Tree View

When using the generated TreeView control program, first import the **tviewdata.imp** file, to populate the required entries in your screenset Data Block.

To use the TreeView control program, simply populate the ATVIEW-PARMS Data Block group with the data you want to insert into the Created Tree control. See your Help for details.

Callback registration and event handler examples are provided in the generated program. You can adapt them to meet your own needs.

10.3.2.4 Toolbar

When using the generated Toolbar control program, first import the **tbardata.imp** file, to populate the required entries in your screenset Data Block.

To use the Toolbar control program, adapt the menu and toolbar button definition used by the program: do this by editing one WORKING-STORAGE copyfile that defines the structure to be used.

Next simply write code to respond to menu/ button choice selection by the user, and take whatever action you require, including execution of existing menu choice dialog which is already defined in your screensets.

10.3.2.5 User Defined

When creating the User control object, you can generate a generic skeleton control program, which provides the structure for you to create any control you choose.

10.3.3 Summary

The generated programs you produce will compile and run successfully without change. You can now change the code to perform any additional functions you require in your application, or, as you have seen, you can use the generated code unchanged.

The intention is to produce a full suite of control programs which provide access to all class library controls via the same procedural COBOL interface. This process will be completed in future releases of Dialog System in Net Express.

11 Multiple Screensets

This chapter shows you how to use multiple programs and screensets using Dialog System. It covers both Dsrunner and using the call interface to maintain multiple screensets in your application.

When using multiple screensets, you can call Dialog System in two ways:

- By using Dsrunner.

  This is the fastest and simplest way to call Dialog System particularly when you are developing a multi-screenset, multi-module Dialog System application.

- By using the call interface using a Router program as described later in this chapter.

11.1 Dsrunner

Most applications should be able to use Dsrunner. This is the simplest way of calling Dialog System. It enables you to develop a modular Dialog System application with multiple screensets and multiple subprogram modules. You can therefore focus on your business logic and supporting screenset, without needing to worry about the details of calling Dialog System.

The Dsrunner program loads the Dsrunner screenset which is a template for a main window that enables you to launch other screensets. It handles any subprogram and screenset switching as required. This means that you do not have to supply any code to switch between multiple screensets or multiple subprograms.

11.1.1 Dsrunner Architecture

Dsrunner is a program which runs your application as a subprogram. This means you use Dsrunner to launch a screenset which involves loading the screenset and calling its associated subprogram.You can launch further screensets from that screenset, providing that you have set up your Data Block correctly.

If you want to use multiple screensets and multiple programs, you need to implement a method of program and screenset switching that ensures that the correct screenset and program are loaded when an event occurs. A sample program, called Router, is provided with Dialog System to demonstrate this and is described later in this chapter.

Dsrunner does all the work that Router does and more.

11.1.2 Dsrunner Operation

Dsrunner is the main program in an application. The application typically also contains other screensets and sub-programs. The main (Dsrunner) screenset provides the first windows in the application and the Dsrunner program provides functions that load other screensets and subprograms.

Each screenset (optionally) has a subprogram associated with it. The association of a screenset with a subprogram is made by using the same name (except for the file extension) for them both. For example, if a screenset is named **dsrnr.gs**, its associated subprogram should be named **dsrnr.cbl**.

When a screenset and its associated subprogram are launched:

1. Dsrunner allocates some memory for the Data Block for the screenset and initializes it to LOW-VALUES.
2. The subprogram is called before the screenset is loaded into Dialog System.

   This allows any initialization to be performed.
3. The subprogram executes an EXIT PROGRAM when it finishes its initialization.
4. The EXIT PROGRAM causes Dsrunner to load the screenset.

5. The screenset executes any SCREENSET-INITIALIZED logic present then enters the event processing loop.

   If a screenset executes a RETC instruction, it returns to Dsrunner.

6. Dsrunner then checks to see whether a Dsrunner function is requested (by checking if a valid SIGNATURE and function code are specified).

   - If a function is requested, Dsrunner performs that function then returns to the screenset.
   - If no function is requested (as would typically be the case), Dsrunner CALLs the sub-program.

7. The sub-program:

   a Performs any requested function.

   b Updates the Data block.

   c Executes an EXIT PROGRAM which returns control to Dsrunner.

   Dsrunner returns control to the screenset.

11.1.2.1 Parameters

When the sub-program is called, it is passed four parameters which must appear in the Linkage Section of the program. They must not appear in the Working-Storage Section of the program because they are parameters for a subprogram.

The parameters are:

- **screenset-data-block** - Mandatory.

  Provided in **screenset.cpb**

  The **screenset-data-block** is the Data Block provided for the screenset. The first time the subprogram is called, this Data Block is initialized to LOW-VALUES. Dsrunner does not perform any Data Block version number checking. You must therefore take great care to ensure that the screenset's Data Block and the subprogram's Data Block are kept synchronized.

- **dsrunner-info-block** - Optional and can be ignored.

  Provided in **dsrunner.cpy**.

- **ds-event-block** - Optional and can be ignored.

  Provided in **dssysinf.cpy.** This is exactly as would be returned by Dsgrun.

- **ds-control-block** - Optional and can be ignored.

  Provided in **ds-cntrl.cpy**. It is used by Dsrunner to call Dsgrun. It contains the values that Dsrunner uses to issue the call to Dsgrun, and also contains values returned by Dsgrun on its return, for example, the error-codes, window names, and object names. The **ds-control-block** is described in detail in the section *The Control Block* in the chapter *Using the Screenset*.

  Your sub-program can modify the **ds-clear-dialog** and the **ds-procedure** fields in **ds-control-block** to change the way the screenset executes. All other input fields are under the control of Dsrunner and cannot be modified.

11.1.2.2 Dsrunner Screensets

Dsrunner can run any screenset without any special actions on your part, providing the **screenset-id** is defined in the **Configuration, Screenset** choice on the **Options** menu. The **screenset-id** is required so that Dsrunner can switch between screensets.

The Dsrunner screenset supplied with Dialog System, **dsrunner.gs** is in the **DialogSystem\bin** subdirectory. This screenset is a template that you can modify. Alternatively you can write your own screenset, but you must make sure that you set up the Data Block correctly and have the required global dialog.

Data Block Header

A Dsrunner screenset must contain the following fields at the start of the Data Block. These fields hold controlling and dispatching information and are:

```
DSRUNNER-DATA-ITEMS       1
  DSRUNNER-SIGNATURE      X 8.0
  DSRUNNER-FUNCTION-CODE  X 4.0
  DSRUNNER-RETURN-CODE    C 2.0
  DSRUNNER-PARAM-NUMERIC  C 4.0
  DSRUNNER-PARAM-STRING   X 256.0
```

- DSRUNNER-SIGNATURE

  This is a signature. If you want to launch screensets in turn from a screenset that has already been launched by Dsrunner, you need to specify a signature in your Data Block that indicates it is a Dsrunner screenset.

- DSRUNNER-FUNCTION-CODE

  This is a function code.

- DSRUNNER-RETURN-CODE

  This is a return code.

- DSRUNNER-PARAM-NUMERIC and DSRUNNER-PARAM-STRING

  These are numeric and string parameters.

If the main screenset does not follow these rules it can still be started. However, no other applications can be launched.

If you want to use a shared memory buffer, the following fields must be inserted immediately after the DSRUNNER-DATA-ITEMS group:

```
SHARED-MEMORY-BUFFER
  YOUR-DATA              any format or size
```

You also need to reserve some fields for Dsrunner's use at the start of the Data Block. For this you can import the file **dsrunner.imp** at the start of the Data Block of the screenset. This import file also contains the key statements required in your global dialog to:

- Set up the Dsrunner signature.
- Enable screenset switching.
- Exit properly.

Dsrunner Screenset Requirements

You must ensure that each screenset to be used with Dsrunner:

- Contains a screenset-id which must be unique for the screensets loaded.
- Uses the following dialog to close it and its subprogram:

  ```
  SET-EXIT-FLAG
    RETC
  ```

This dialog can be in global or local dialog and you can attach it to any suitable event. If you want to tell the subprogram that it is closing down, set your own termination flag and do a RETC before you set the exit flag and do the RETC for Dsrunner.

- Contains the following global dialog if you want to handle multiple screensets:

  ```
  OTHER-SCREENSET
    REPEAT-EVENT
    RETC
  ```

Dsrunner Global Dialog

This dialog describes the key parts of the global dialog in the supplied Dsrunner screenset:

```
OTHER-SCREENSET
  REPEAT-EVENT
  RETC
```

This dialog causes an event that occurs for an inactive screenset to be repeated (stacked for the inactive screenset). The RETC causes Dsrunner to switch the active screenset to the correct screenset. This is required if you want to control more than one screenset.

```
SCREENSET-INITIALIZED
  MOVE "DSRUNNER" DSRUNNER-SIGNATURE(1)
```

This dialog sets up the signature indicating that the Data Block of this screenset is set up for Dsrunner. If you do not set up the Data Block, Dsrunner ignores any function codes and a RETC invokes only the subprogram associated with the screenset.

```
CLOSED-WINDOW
   EXECUTE-PROCEDURE  EXIT-PROGRAM
CLOSEDOWN
   EXECUTE-PROCEDURE  EXIT-PROGRAM
  EXIT-PROGRAM
   SET-EXIT-FLAG
   RETC
```

On a request to close the application, the SET-EXIT-FLAG is issued. This causes Dsrunner to terminate after the RETC. If you want to perform termination processing in your subprogram, set your own termination flag and do a RETC before you set the exit flag and do the RETC to Dsrunner.

```
 OPEN-SCREENSET
   MOVE "file" DSRUNNER-FUNCTION-CODE(1)
   MOVE "*.gs" DSRUNNER-PARAM-STRING(1)
   RETC
   IFNOT= DSRUNNER-RETURN-CODE(1) 0 OPEN-SCREENSET-ERROR
* resulting file name is left in param-string
   MOVE "lnch" DSRUNNER-FUNCTION-CODE(1)
   RETC
 OPEN-SCREENSET-ERROR
```

This dialog shows how to perform a Dsrunner function. In this example, the file requester is shown and a file name obtained. The function to launch a screenset is then executed.

11.1.3 Dsrunner Program and Functions

The Dsrunner program provides certain functions through the first few fields of its Data Block. These include:

- Starting and stopping new applications (screensets) or instances of applications.
- Switching into Dialog System trace mode for debugging purposes.
- Creating and using shared memory areas.

For a list of all of the available functions, see the topic *Dsrunner Functions* in the Help.

11.1.4 Using Dsrunner Functions

Because Dialog System is called by Dsrunner, you set the Dsrunner function code in the Dsrunner Data Block before executing a RETC from the Dsrunner screenset. For example:

```
LAUNCH-SCREENSET
  MOVE "lnch" DSRUNNER-FUNCTION-CODE(1)
  MOVE "screenset-name" DSRUNNER-PARAM-STRING(1)
  RETC
  MOVE DSRUNNER-PARAM-NUMERIC(1) SAVED-SS-INSTANCE
```

In this example, `screenset-name` is the name of the screenset to launch. The function returns the screenset instance, which is saved as `SAVED-SS-INSTANCE`.

11.1.5 Starting Screensets Using a Command Line

Dsrunner is designed to be run from a command line:

```
runw dsrunner [screenset-name /l /d screenset-name]
```

Where:

screenset-name is the name of the initial screenset to load. You can enter it in the first positional parameter, or following the /l "load screenset" parameter. You can also specify the /d switch.

/d enables the Screenset Animator immediately, so the Screenset Animator is invoked when the first line of dialog in the first screenset is executed.

Dsrunner is provided in both **.obj** and **.gnt** formats to enable you to package your application appropriately for the run-time environment you are using.

11.1.6 Starting Screensets in Net Express IDE

Screensets can also be started via Dsrunner in the Net Express IDE:

1. Select **Settings** on the **Animate** menu.
2. Specify **dsrunner** in **Start animating at**.
3. Specify the options listed above (/l, /d) in **Command line parameters**.

Any program breakpoints you have will stop execution and load the relevant program into a debug/edit window.

11.1.7 Starting a Screenset from a Program

Instead of using a command line, you can call Dsrunner from a program. This enables you to choose your own name for your application and to specify the initial screenset name, without specifying it on the command line. This is useful if your application takes its own command line arguments. It is also useful to perform any application-specific initialization, for example opening a library.

11.1.8 Launching a Screenset

To launch a screenset:

1. Select **Open** on the **File** menu in the Dsrunner window.
2. Select a screenset and click **OK**.

   The screenset you select is loaded and the associated program run.

You can select another screenset and run that by repeating the above instructions.

11.1.9 Launching an Application

In this section we look at the Dsrunner architecture and consider what you need to do to be able to use Dsrunner effectively. You can modify the Dsrunner screenset to meet your own requirements. The supplied screenset is only an example. You can change the Data Block, but do not make changes that affect `dsrunner-info-block`.

When you launch a screenset, Dsrunner goes through the following procedure:

1. When you select **Open** on the **File** menu in the Dsrunner window, you are prompted to enter a screenset name.
2. Dsrunner checks that the screenset is correct, that is, that you have specified a screenset-id. If the screenset is:

   - Not correct, a message box is shown with appropriate text.

- Correct, enough memory is obtained for a dynamically allocated Data Block and is initialized with LOW-VALUES.

3 The program with the same root file name (and path) as the screenset name is called.

   This program can be any valid COBOL executable in the current directory or $COBDIR (normal COBOL program search rules apply). If the program is not found, a message box is shown with appropriate text.

4 A call is made to the subprogram to enable the subprogram to perform any initialization required, including the initialization of the Data Block.

   The following parameters are passed:

   - `Data-Block`
   - `Dsrunner-Info-Block`

     This is provided in **dsrunner.cpy** and contains:

     - `screenset-id`
     - `ds-session-id`
     - `screenset-instance-number`
     - Error codes
   - `Ds-Event-Block`

5 Once the subprogram is initialized, it must return to Dsrunner with a return code of zero:

   - If it returns with a non-zero return code, Dsrunner displays a message box and signals the error.
   - If it returns with a return code of zero, Dsrunner loads the screenset (screenset initialization occurs).

     If an error occurs in the call to Dialog System to load the screenset, the subprogram is called with error-codes specified in the Dsrunner information block.

6 Whenever the screenset executes a RETC, the subprogram will be called.

7 To close down the application, the screenset must SET-EXIT-FLAG in the dialog.

11.1.9.1 Running the Sample Subprogram

Dsrnr is a sample subprogram supplied with Dialog System. To find out how to launch it and for details of key sections of the sample code, see the chapter *Sample Programs*.

11.2 Multiple Screensets and the Router Program

The Dialog System run-time system can be programmed to enable the use of:

- Multiple screensets.
- Multiple instances of the same screenset.

Using these features you can:

- Divide your user interface into logical components.
- Use multiple copies of the same screenset.
- Group all your error messages into a single file.

When you design your screenset and calling program, you should consider in detail how you control screenset handling, looking at issues such as:

- How to divide functions and whether they should be grouped into separate screensets.
- Whether to provide different screensets to groups of users with different access to data, as a security consideration.
- Whether to use multiple instances of a screenset so that a user could display, compare or edit data at the same time.

For more information on the basics of screen control using the Dialog System call interface, see the topic *The Call Interface* in the Help.

11.2.1 Using Multiple Screensets

You can use multiple screensets by pushing and popping them from the screenset stack. By definition, this is a first in, last out operation. Pushing and popping screensets is useful to:

- Remove a screenset used for a particular function from the display when it is no longer required.
- Load multiple screensets during program initialization, and push and pop (or use) them when you need them.
- Keep the display uncluttered by windows that are not being used or do not have input focus.

There are no pre-conditions for pushing a screenset onto the screenset stack, and any screenset, or occurrence of a screenset, can be pushed or popped. Pushed screensets are normally stacked in memory, but if memory is short they will be paged to disk.

11.2.2 Using Multiple Programs and Screensets

The recommended way of developing a large application is to build a separate module for each component and associate each module with its own screenset. Each component is then a separate COBOL program complete with its own user interface.

For example, if you were building an application that has a main data entry component and two utility components, one utility would handle all the printing functions and the other handle the file management functions.

Several screensets can be used in the same application, each being placed on the screenset **stack**. This stack is similar in concept to the call stack that is used whenever you call a program. Calling screensets in turn is easy, because you simply make the call to Dialog System with a value of **N** in `ds-control` and with the new screenset name in `ds-`

set-name. By default, the old set is cleared from the screen before a new one is started.

Once you have multiple programs, each with an associated screenset, you need a method to ensure that the correct program and the correct screenset are made active when an event occurs. The sample program that follows, Router, shows you how to structure an application to have multiple programs, each with an associated screenset.

11.2.3 Terms and Concepts

Before you look at the Router program, you need to understand a few terms and concepts.

11.2.3.1 The Active Screenset

When several screensets are loaded, we need to distinguish between the active screenset and inactive screensets. Graphical objects displayed by screensets that are presently inactive do not differ in appearance from those displayed by the active screenset. The active screenset is the one that is currently receiving events. Only one screenset can receive events at a particular instant.

When using multiple screensets, you usually specify the active screenset by calling Dialog System and specifying ds-use-set in ds-control. The screenset specified in ds-set-name becomes the active screenset. This call is very fast and any overhead resulting from this swapping is minimal.

11.2.3.2 Events for Other Screensets

If an event occurs for a screenset other than the one that is currently active, a special event, the OTHER-SCREENSET event, occurs in the current screenset. This event simply tells the current screenset that an event has occurred that should be posted to another screenset. Any resulting action depends on the logic in the active screenset.

If the OTHER-SCREENSET event is not found (usually in global dialog), nothing happens. You must specify the OTHER-SCREENSET event so a screenset can detect that an event has occurred for another screenset.

Once such an event is detected, you can then make the appropriate screenset active. The most common way is to set a flag indicating that the OTHER-SCREENSET event has occurred and return to the calling program to change screensets.

All events that occur for inactive screensets are returned to the active screenset as OTHER-SCREENSET events.

You can identify the correct screenset by calling Dialog System and specifying the `ds-event-block`. When Dialog System returns to the calling program, the screenset-id of the screenset that the event was really for is in `ds-event-screenset-id`. This screenset-id is specified in **Configuration, Screenset** on the **Options** menu. It is not mandatory to specify a screenset-id, but if you do not, you cannot use multiple screensets because you cannot differentiate between the screensets on the stack.

When the correct screenset is active, the original event is repeated providing that you have specified the REPEAT-EVENT dialog function within the OTHER-SCREENSET event.

Note: The repeated event is not OTHER-SCREENSET: it is the event that would have occurred if the correct screenset had been active.

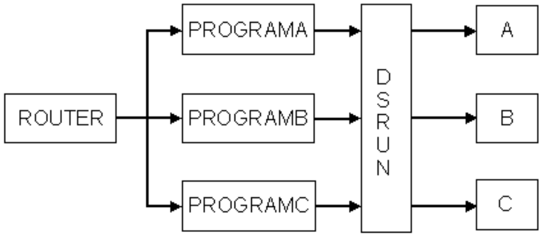
11.2.4 Multiple Screenset Sample Application Using Router

A sample application illustrating how to use Router to handle multiple screensets in Dialog System is included in your sample directory. In this example, Programa, the main program, calls subprograms Programb and Programc. Each program has its own screenset.

A fourth program, Router, handles the routing function. Its only purpose is to determine which program (and screenset) needs to be executed next and then call that program.

Figure 11-1 shows the application structure.

Figure 11-1. Structure of Router Application



11.2.5 Using Multiple Instances of Screensets

As well as using multiple screensets, you can use multiple instances of the same screenset. You use multiple instances in a very similar way to using multiple screensets. The real difference is in identifying the screenset. See the section *Using Multiple Programs and Screensets* before you read this section.

Possible uses of multiple instances of screensets are:

- A screenset containing one window with its associated controls.
- Working with data groups where each group item has the same format. Using multiple instances of the same screenset, you can have one screenset to display, compare or update the group items as required.

Using multiple instances of the same screenset requires your program to:

- Track the number of instances of a screenset.
- Ensure that the Data Block being passed to Dsgrun is the correct one for that screenset instance.

When multiple instances are used, a screenset is first started by an "N" or "S" call.

To create a new instance:

1 Push a new screenset onto the stack by calling Dialog System.

2 Specify `ds-push-set` in `ds-control`.

3 When Dialog System returns, it places an allocated instance value in the `ds-screenset-instance` Control Block field.

   An instance value is always returned, but you only need to use it when you use multiple instances of a screenset.

The instance value is unique to that particular screenset instance. Your application must keep track of instance values because they are not assigned in any particular order.

11.2.5.1 Tracking the Active Instance Value

You can track the active instance by examining `ds-event-screenset-id` and `ds-event-screenset-instance-no` within **dssysinf.cpy**, which must be copied into your program Working-Storage Section. For more information on **dssysinf.cpy**, see the chapter *Using Panels V2*.

To specify that you want a new instance of a screenset to be loaded:

- Set `ds-control` to `ds-use-instance-set` when you call Dsgrun.

To identify the appropriate instance for a particular event:

- Examine `ds-event-screenset-instance`. This contains the appropriate instance value.

To call Dialog System with a particular instance value:

1 Move `ds-event-screenset-instance-no` into `ds-instance`.

2 Specify the required screenset name by moving `ds-event-screenset-id` into `ds-set-name`.

3 Call Dialog System.

Notes:

- To use multiple instances of a screenset, you must set the screenset-id to be the name of the screenset using **Configuration, Screenset** on the **Options** menu.
- There is no support for using multiple instances of a screenset when running through the Screenset Animator, using the definition software. If, however, Dsgrun is called from the application then multiple instances are supported. For more information on the Screenset Animator see the section *Screenset Animator* in the help.

11.2.5.2 Using the Correct Data Block

If you are using multiple instances of a screenset, you must maintain multiple copies of the Data Block. You can do this in several ways:

- If you know the number of instances that will be created, the easiest way is to use code similar to the following:

  ```
  copy "program.cpb"
  ```

  replacing `data-block-a` by `data-block-b`.

- Implement a Data Block heap or stack.

  To push a screenset onto the screenset stack and start a new screenset, call Dsgrun using the following:

  ```
      move ds-push-set to ds-control
      call "dsgrun" using ds-control-block,
                          data-block
  ```

  Where `ds-push-set` places the value `"S"` in `ds-control`. The existing screenset is pushed onto the screenset stack.

  When you pop a screenset off the screenset stack you can use either of the following:

  - `ds-quit-set`

    This closes the existing screenset and pops the first screenset off the top of the screenset stack.

- `ds-use-set`

  This pops the specified screenset off the screenset stack without closing the existing screenset.

For more information see the topic *Screenset Animator* in the Help.

11.2.5.3 Sample Programs for Multiple Instances

There are two sample programs which demonstrate the use of the call interface by your program:

- **push-pop.cbl** demonstrates the use of screenset pushing and popping.
- **custom1.cbl** demonstrates the use of multiple instances of the same screenset.

You can find key sections of code for these programs in the chapter *Sample Programs*.

11.2.6 The Router Program

Before we look at the Router program logic, let's take a look at the copyfile that represents a data area shared by Router and all the programs that are called by Router.

The following code shows the copyfile **router.cpy**, which contains a program name and two flags.

`program-name` contains the name of the program to be called next.

`cancel-on-return` is set to TRUE by a program which is to be cancelled when it returns to Router. If the main program (Programa) requests cancel, Router sets `exit-on-return`, then exits the main perform loop.

```
01  program-control.

    03  dispatch-flag          pic 9(2) comp-5.
        88 cancel-on-return    value 1 false 0.

```

```
     03  exit-flag              pic 9(2) comp-5.
         88 exit-on-return      value 1 false 0.

     03  program-name           pic X(8).
```

Router starts by calling the main program Programa. When Programa determines that a sub-program is required to handle a particular function, it puts the name of the sub-program in `program-name` and exits. Router then calls the program in `program-name`.

```
 main-section.

*    Make sure we don’t exit straight away
     initialize exit-flag

*    Start by calling the main program
     move main-program to program-name

*    Call program in program-name until exit is requested
     perform until exit-on-return

*      Remember who we’ve called
       move program-name to dispatched-program

*      Dispatch program in Program-Name
       call program-name using
       if cancel-on-return
*        If last program requested cancel - do cancel
         set cancel-on-return to false
         cancel dispatched-program
         if dispatched-program not = main-program
*          Re-load main program if sub-program cancelled
           move main-program to program-name
         else
*          If main program requested cancel, request exit
           set exit-on-return to true
         end-if
       end-if
     end-perform

     stop run.
```

If you want to cancel a program, the program must set `cancel-on-return` before it returns to Router. If the program specified in `main-program` requests cancel, Router exits after cancelling the main program.

11.2.7 The Main Program

This section shows the source for the main program that the Router calls, Programa. The two subprograms Programb and Programc are very similar.

Lines 31-44:

```
31 procedure division using program-control.
32
33 main-section.
34     if new-instance
35*    First time in we push a new screenset onto the stack
36          perform new-set-instance
37     else
38*    Once we’ve initialized use existing screenset
39          perform use-set-instance
40     end-if
41*    Call Dialog System as long as we should be active
42     perform until program-name not = this-program-name
43          perform call-ds
44     exit-program.
```

The main section of Programa checks whether the screenset instance has been created. If it has, the program uses it; if it hasn’t, the program creates it.

Once an instance exists, Dialog System is called to display it. Dialog System is called until the name of the program in `program-name` is changed from Programa.

Programa exits (and returns to Router) if `program-name` is not Programa. Router, in turn, calls the program in `program-name`.

Lines 46-60:

```
46 new-set-instance
47 ...
58 ...
59*    Push a new screenset onto the stack
60     move ds-push-set to ds-control.
```

The most important part of `new-set-instance` is specifying `ds-push-set` so that when Dialog System is called. it places the screenset on the stack.

Lines 62-66:

```
62 use-set-instance.
63*    Use existing screenset on the stack
64     move ds-use-set to ds-control
```

```
 65*    This is what we’re called
 66     move this-program-name to ds-set-name.
```

When the screenset instance has been created, we can use it by telling Dialog System the screenset name and directing Dialog System to use that screenset from the stack.

Lines 67-101:

```
 67 ...
 68 call-ds.
 69* Standard initialization and call
 70     initialize programa-flags
 71     call "dsgrun" using ds-control-block
 72                         programa-data-block
 73                         ds-event-block
 74     evaluate true
 75* When this Screenset flag is set we tell router to exit
 76      when programa-terminate-true
 77         set exit-on-return to true
 78 ...
 87 ...
 88      when programa-other-set-true
 89         move ds-event-screenset-id to program-name
 90* If the Programb menu item is selected, this flag is set
 91      when programa-program-b-true
 92*       So we then request Programb to be dispatched
 93         move "programb" to program-name
 94* If the Programc menu item is selected, this flag is set
 95      when programa-program-c-true
 96*       So we then request Programc to be dispatched
 97         move "programc" to program-name
 98* Its an event for us
 99      when other
100         move "Hello A" to programa-field1
101     end-evaluate.
```

This code fragment shows a standard call to Dialog System specifying all three parameters, followed by an evaluation. This evaluation directs the overall operation of the program, based on flags set in screenset dialog.

Depending on the flags set in the screenset dialog, Programa exits, switches screenset and program, calls a subprogram, or handles an event for it.

The crucial element of this code is the switching of programs and screensets when the `OTHER-SET-TRUE` flag is set. Dialog System places the Screenset-ID (set using **Configuration, Screenset** on the **Options** menu) of the screenset for which the event occurred in ds-event-screenset-id.

This means that Programa can detect the correct screenset-id and hence program name because this example, conveniently, calls the screenset and its associated program by the same filename. If the screenset and its associated program name differed, you would need to identify the screenset then call the appropriate program.

Programa puts the screenset-id into ds-screenset-name and exits; Router will call the identified program and hence load the correct screenset.

11.2.8 Multiple Screenset Dialog

The following example dialog shows screenset dialog needed to handle multiple screensets. The dialog listed here is taken from the Programa screenset, but it is the same for the other screensets.

Dialog System uses the OTHER-SCREENSET event to detect that an event has occurred in a screenset other than the one that is currently loaded. In this example, this event is located in the global dialog table for each screenset:

```
OTHER-SCREENSET
    SET-FLAG OTHER-SET(1)
    REPEAT-EVENT
    RETC
```

When such an event is detected, the flag OTHER-SET is set, to signal to the program that another screenset is to be used.

Next, the REPEAT-EVENT function is executed. REPEAT-EVENT tells Dialog System to repeat the last event the next time it goes for input. This occurs when the next screenset is loaded and control is in the next program. After the REPEAT-EVENT, the screenset returns to the calling program using RETC .

The calling program, in this case Programa, can then check the screenset flag. If the flag is set, the program can place the screenset-id in ds-set-name (assuming the screenset name and program name are the same) and exit to Router. Router then calls the appropriate program and hence loads the correct screenset. Then, because of the REPEAT-

EVENT function issued by the Programa screenset, the event that caused the OTHER-SCREENSET event is repeated.

Note: The event repeated is not OTHER-SCREENSET; it is the event that would have occurred if the correct screenset had been active.

For more information on these functions, refer to the topic *Dialog Statements: Functions* in the Help.

11.2.9 The Sequence of Events

The previous sections have described the overall process of switching screensets and the key events that occur. However, if you trace the program flow, all is not quite so simple in practice. There are, in fact, several program switches before control is left in the appropriate program.

Why? The key reason is the change of focus. When the input focus moves from one graphical object to another, two events occur. The object losing focus receives the LOST-FOCUS event, and the object gaining focus receives the GAINED-FOCUS event.

If the object that loses focus is a control object, the LOST-FOCUS event also occurs for the window that contains that object. Similarly, if the object that gains focus is a control object, the GAINED-FOCUS event also occurs for the window that contains the object.

This table shows the sequence of events that actually occurs when there is a simple change of focus from A's window to B's window.

System Event	Active Screenset	Event Screenset	DS Event
Mouse-button-1-down	A	B	OTHER-SCREENSET
Lost-focus (window A)	B	A	OTHER-SCREENSET
Gained-focus (window B)	A	B	OTHER-SCREENSET
Mouse-button-1-up	B	B	ANY-OTHER-EVENT

Each screenset traps the OTHER-SCREENSET event and returns to the calling program from Dialog System. Dialog System is then called with the correct screenset (identified from `ds-event-screenset-id`).

Pressing the mouse button on B's window when A has the focus causes the sequence of events shown in this table.

The system event Mouse-button-1-down on window B causes Dialog System to present an OTHER-SCREENSET event to the active screenset. This gives program A a chance to switch to screenset B.

However, the next system event is window B losing focus. But, as you can see in the table, this event occurs only after the switch to B. This in turn causes an OTHER-SCREENSET event in B.

Another switch of screensets results. B now gains focus when A is active, so another OTHER-SCREENSET event results. This causes B to become active. The mouse button up event is now presented to B as ANY-OTHER-EVENT (because all mouse events are translated to this event).

This sequence of events need not trouble you once you understand the principle of screenset swapping because no matter how many screenset swaps take place, the correct screenset is left active once a steady state is reached.

Note: In this example, the mouse button down event is lost. You can rectify this using the REPEAT-EVENT function within the OTHER-SCREENSET dialog.

11.2.9.1 Repeating the Event

First, you must decide whether you need to repeat the event. The purpose of repeating the event is clear. If you need the event, it must be repeated in the correct screenset. However, in many cases, you do not need the event, therefore you do not need to repeat it.

The event sequence caused by clicking buttonB on windowB when focus is currently on buttonA on windowA is the following:

```
Mouse-Button-1-Down
Mouse-Button-1-Up
Lost-Focus on buttonA
Lost-Focus on windowA
Gained-Focus on windowB
Gained-Focus on buttonB
Button-Clicked on buttonB
```

This sequence shows that you do not need REPEAT-EVENT in this situation, because the Gained-Focus on windowB loads ScreensetB **before** the Button-Clicked event causes a BUTTON-SELECTED event.

11.2.10 Setting the Focus

When a screenset is made active by calling Dialog System and specifying `ds-use-set` in `ds-control`, it does not automatically receive the focus. If the screenset switch is caused by the user (by selecting a new window or control), the focus moves to the new screenset because the user's actions cause a GAINED-FOCUS event.

If the switch is caused by the program, you must specify a dialog procedure name in `ds-procedure` before you call Dialog System. Use a SET-FOCUS dialog function to set the focus on the appropriate window.

11.3 Further Information

For a detailed description of the call interface see the topic *The Call Interface* in the Help, which provides information on the Control Block, including the Event Block, the Data Block, use of the Screenset Animator, version checking, and values the calling program returns to Dialog System.

12 Migrating to Different Platforms

This version of Dialog System is primarily for Windows 95 and Windows NT.

This chapter explains the major differences between these environments and gives you guidelines for handling screensets that are used in more than one environment.

You can run a screenset on other operating systems if you have the appropriate earlier version of Dialog System, because the system uses the appropriate run-time software (it is not the responsibility of the calling program). Some object definitions are not portable across environments. Dialog System provides optional portability warnings. If enabled, **Portability warnings** on the **Options, Include** menu causes Dialog System to display a warning message whenever you are about to create an object that is not portable.

When dealing with portability issues, we recommend that you test your application thoroughly, as Dialog System cannot guarantee to detect all portability violations because of the way the environment is constantly changing.

12.1 Differences Across Environments

Superficially, many of the objects and controls look the same on any of the supported environments. Objects such as radio buttons and check boxes provide the most obvious differences between Presentation Manager and Windows.

The objects that Dialog System creates follow the appearance of similar objects in the current environment. Therefore, in applications created under Presentation Manager, radio buttons appear as outlined ovals that are filled with a color when selected. As you move the screenset from one environment to another, the appearance of the objects changes.

The behavior of the mouse varies between the environments but can be fixed by the design of the application.

There are some objects and menu choices that are available on only some environments. For example, the OLE2 object is available only under Windows.

If you want an application to be portable, you must use the functions and facilities that are common among the environments you want to run on, so that the application has a common look and feel on all environments, rather than taking advantage of any one environment's functions. If you don't need a portable application, you can design just for the one environment and take full advantage of anything specific to that environment.

For example, if you want to exploit the 32-bit Windows interface, you can use the class library of your COBOL system to extend your Dialog System interface. However the interface will not be portable to any earlier versions of Dialog System.

12.1.1 Desktop Mode

One major difference among the environments is how Dialog System starts up.

Presentation Manager: Under Presentation Manager, Dialog System starts up with the Dialog System window. Every window you create is created as a clipped window within the Dialog System window, unless you switch to Desktop mode. In Desktop mode, you create objects directly on the desktop.

Windows: Under Windows, Dialog System starts up in Desktop mode. This is because Windows does not permit the creation of clipped windows with menus. Obviously, this causes a problem because the first window you need to create would be clipped if it was within the Dialog System window. Creating windows directly on the desktop avoids this problem.

Windows: Under Windows you must be in Desktop mode to see a menu bar on a window. However, you can edit a menu bar even if you are not in Desktop mode.

12.2 Developing for Graphical and GUI Emulation Environments

GUI emulation is supported in 16-bit versions of Dialog System. For information on creating screensets that are portable for GUI emulation, see your 16-bit product documentation.

12.3 General Portability Guidelines

These guidelines are primarily for portability between graphical interfaces. If your screenset is to be portable to a GUI emulation environment, consider the guidelines given in the section *Developing for Graphical and GUI Emulation Environments* in your 16-bit product documentation.

- For cross-resolution portability, you should bear in mind that a lower resolution environment cannot display the same amount of information as a higher resolution environment. For example, you can display more information on an XGA screen than you can on a VGA screen. For this reason, you should define screensets on the lowest resolution available. This way, you will avoid the possibility of application windows being clipped by the right/bottom edges of the screen.

  Dialog System does however provide multiple resolution support for run-time use. See the topics *Calls to Dsgrun* and *The Call Interface* in the Help, and the chapter *Advanced Topics*.

- Be aware of the amount of screen space available. For example, a 43-line GUI emulation screen can hold more information than a standard VGA resolution, but a 25-line screen cannot.

- Not all fonts are available in all environments. Only the default font, the system monospaced font, and the system proportional font are guaranteed to be available in all graphical environments. (GUI emulation does not support multiple fonts; use system monospaced font in this mode.)

- System proportional fonts are different in different environments. This affects both static text, and text defined within objects such as buttons. Buttons are particularly important because initially they are defined to fit the text inside them. This guarantees the visibility of the button text, regardless of the environment in which the buttons were created.

  However, this does mean that the width of the buttons varies to accommodate the text. This is why, for example, closely horizontally spaced buttons under VGA can cause horizontal overlap when viewed under XGA. Sizing a button will remove this feature and avoid the overlap. However, the button text may be clipped. Changing to a typeface with a particular point size (rather than using the system proportional font) may help.

  The default border of a default push button is drawn around the defined area of the button. Some environments emphasize this more than others. Again, the solution is simply to avoid close spacing of objects.

- Use **Fit-Text** (on button properties dialog boxes) consistently. For each button group, do not mix the use of **Fit-Text**. Select this property for all buttons in the group, or for none of them.

  For example, consider two push buttons containing identical text. Both initially have **Fit-Text** selected. If you now de-select **Fit-Text** for one of the buttons, you will not notice any difference in the size of either button. They will both have the same dimensions. Now, if you save this definition and load it on another environment, you will almost certainly notice a difference in size. In fact, depending on the text you have chosen, one button may actually clip the text.

- Portability of Fonts. Fonts that have been assigned style names are portable among environments. Any font with a style name can be remapped at run time to use a completely different font. To do this mapping, Dialog System looks for a binary file containing mapping information. This is the font side file. See the topic *Font Side files* in the Help for more information.

- For portability to GUI emulation, select **Emulation** on the Alignment dialog box. You can set this option as a default in **ds.cfg**. See the topic *Dialog System Overview* in the Help.

12.4 Other Cross Environment Issues

If you are creating a screenset for use on all three graphical environments, you should select **Portability Warnings** on the **Options, Include** menu, so that you are aware of the implications when you create particular types of objects.

When a warning is displayed, you can continue to create the object that caused it (and ignore the warning), but if you intend to run the screenset on the environment mentioned in the warning, the results will not be as you intend. In most cases, you will be able to redesign that part of the screenset to use objects that are supported on all the target environments.

In particular, note the following:

- Clipped child windows cannot have menu bars under Windows. Presentation Manager will allow this.
- Non-clipped windows will not be unshown when the parent window is unshown under Windows. Write Dialog to take this into account.
- Non-clipped windows should have a title bar under Windows.
- Center and Right justification is not available under Windows.
- Setting color on push buttons or scroll bars has no effect under Windows.
- Under Windows, windows with title bars must have a border.
- Message boxes are always movable under Windows.
- Read-only multiple line entry fields are not supported under Windows 3.0, but they are under Windows 3.1.
- Push buttons without borders are not supported under Windows.

12.5 Backward Compatibility Issues

There are two issues here:

- Notebooks.
- Containers.

12.5.1 Notebooks

The tab control object replaces the notebook control object that was available in previous versions of Dialog System. The functionality of the tab control is similar to that of the notebook, but the following features that were available with the notebook control are no longer available with the tab control:

- Bindings.

  The backpage intersection and binding properties of the notebook are ignored.

- Minor tabs.

  Minor tabs are converted into major tabs.

- Pages with no tabs.

  All tab control pages have tabs.

- Tab text alignment.

  The bitmaps and text displayed in tabs are always centered within the tab.

- Tab shape.

  The shape of tab control page tabs is fixed.

- Status line text.

  Tab controls do not have status lines.

- Tab shape.

  Tab control tabs are always rectangular.

In addition, the base release of Windows 95 does not support tab orientation. However, an upgrade to the operating system is included in Microsoft Internet Explorer 3.0 and later which provides this support. Also, note that unlike with notebooks, display corruption may result if the tab text or bitmap does not fit completely within the tab.

Dialog System will preserve the properties of notebook objects created using previous versions of Dialog System. However, those properties that are no longer supported cannot be edited from within the definition software.

12.5.2 Containers

The container implementation in Dialog System uses a Windows control called a List View. This control provides a subset of the functionality that was available with the container object.

The main limitation of the List View control is that the first column in the list view must contain only icons, and the remaining columns must contain only text.

In addition, the following features of the container are no longer supported:

- Overall title.

  There is no overall title in a List View. If an overall title is specified, it will be ignored.

- The formatting of the list view columns is defined by Windows.

  All flags used to control the formatting of column headings or data is ignored, including alignment, separator, and column width settings.

- Delta events are not supported.

  Setting of delta events will be ignored and no delta events will be generated.

- Columns cannot be hidden once created.

  The Sc and Hc functions of the Dscnr Dialog System extension have no effect.

- Fonts cannot be set.

If you intend to implement any new container objects in your application, consider using the List View object in the GUI class library, as it provides more control over the list view object.

12.6 Compatibility Chart

	Environment:				
	Presentation Manager	**Windows**	**GUI Emulation**	**Windows NT and 95 (16-bit)**	**Windows NT and 95 (32-bit)**
3-D objects	x	y	x	y	y
Bitmaps	y	y	y	y	y
Check box	y	y	y	y	y
Color	y(9)	y(9)	x	y(9)	y(9)
Container	y	y	x	y	x(11)
Dialog box	y	y	y	y	y
Entry field	y	y	y	y	y
Group box	y	y	y	y	y
Fonts	y(8)	y(8)	x	y(8)	y(8)
Icon/Bitmapped buttons	y	y	x	y	y
List box	y	y	y	y	y
List view	x	x	x	x	y
Menu bar	y	y	y	y	y
Message box	y	y (4)	y	y (4)	y (4)
Multiple line entry field	y	y (5)	y	y	y
Primary window	y (1)	y (1)	y (1)	y (1)	y (1)
Push button	y	y (6)	y	y (6)	y (6)
Radio button	y	y	y	y	y
Scroll bars	y	y	y	y	y
Secondary window (clipped)	y	y (3)	y	y (3)	y (3)

	Environment:				
	Presentation Manager	**Windows**	**GUI Emulation**	**Windows NT and 95 (16-bit)**	**Windows NT and 95 (32-bit)**
Secondary window (not clipped)	y	y(2)	y	y (2)	y (2)
Selection box	y	y	y	y	y
Tab Control	y	y	x	y	x(10)
Tab Control Page	x	x	x	x	y(10)
Text	y	y	y(7)	y	y
User control	x	x	x	x	y

Notes:

- **System position** has no effect.
- The following colors are common across all environments: Black, Blue, Brown, Cyan, Green, Magenta, Red, White, Yellow.

13 Using Panels V2

Dialog System has Panels Version 2 (Panels V2) as its underlying technology.

Using Panels V2 with Dialog System enables you to:

- Maintain finer control of objects than with Dialog System alone.

  For example, using Panels V2 calls, you can switch off the System Menu.

- Access Panels V2 features that Dialog System does not support.

  For example, the rubber banding techniques that you see in the Definition Facility software do not exist in Dialog System but are available in Panels V2.

This chapter shows how you can enhance your Dialog System application with Panels V2.

13.1 Calling Panels V2

The following fragment is an example of a Panels V2 call statement.

```
call "PANELS2" using p2-parameter-block
                     p2d-dialog-box-record
                     new-title-buffer
                     attribute-buffer
```

Panels V2 calls Pan2win, and Pan2win issues the appropriate Windows API calls.

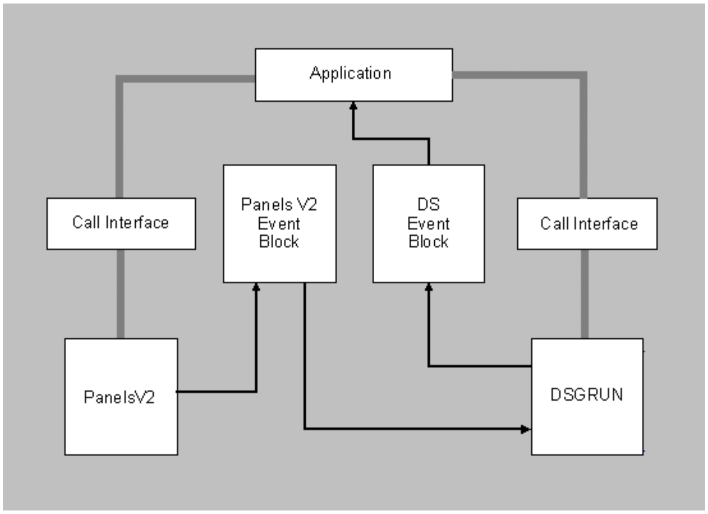
The set of functions that Panels V2 provides are the same regardless of the environment you are using, assuming of course that the environment supports the display feature.

Dialog System is based on this technology.

13.2 Dialog System and Panels V2 Events

Figure 13-1 shows a typical Dialog System application that contains calls to Panels V2.

Figure 13-1. Dialog System/Panels&nbsp;V2 Events



Notice how events are handled. There is no direct line for events between Panels V2 and your application. When an event occurs, Panels V2 detects the event and returns the event information through Dialog System (Dsgrun). Dialog System then returns the event information to your program through the Dialog System Event Block (**dssysinf.cpy**). You must use dialog to determine if (and how) you want to handle the event in Dialog System or your application.

13.3 Copyfiles

You must include several copyfiles in the Working-Storage Section of your program. For example:

```
 working-storage section.

     copy "ds-cntrl.mf".
     copy "clip.cpb".
     copy "pan2link.cpy".
     copy "dssysinf.cpy".
```

The first two you are already familiar with: the Dialog System Control Block and the Data Block generated from your screenset. See the topic *The Call Interface* in the Help for a description of these files.

The other two copyfiles are described in the next two sections.

13.3.1 Panels V2 Copyfile (pan2link.cpy)

Your COBOL program also should contain the **pan2link.cpy** copyfile .

The file contains:

- Record definitions.
- Predefined level-78 parameter definitions.
- A Micro Focus key list.

Although this file is not absolutely necessary, the definitions in this file greatly simplify the writing of the Panels V2 part of the application.

Browse this file to get an idea of the type of information it contains.

13.3.2 Dialog System Event Block (dssysinf.cpy)

This file contains the definitions of the Dialog System Event Block. When an event occurs, information about the event is passed through the Panels V2 Event Block back to Dialog System. (Refer to Figure 13-1.) **dssysinf.cpy** is just a copy of the Panels V2 Event Block with extra fields added on for Dialog System use.

```
01 ds-event-block.
78 ds-eb-start        value next.
   03 ds-ancestor                       pic 9(9)  comp-5.
   03 ds-descendant                     pic 9(9)  comp-5.
   03 ds-screenset-id                   pic x(8).
   03 ds-event-screenset-details.
      05 ds-event-screenset-id          pic x(8).
      05 ds-event-screenset-instance-no pic 9(2)  comp-x.
   03 ds-event-reserved                 pic x(11).
   03 ds-event-type                     pic 9(4)  comp-5.
   03 ds-event-data.
      05 ds-gadget-event-data.
         07 ds-gadget-type              pic 9(2)  comp-x.
         07 ds-gadget-command           pic 9(2)  comp-x.
         07 ds-gadget-id                pic 9(4)  comp-5.
         07 ds-gadget-return            pic 9(4)  comp-5.
      05 ds-mouse-event-data.
         07 ds-mouse-x                  pic s9(4) comp-5.
         07 ds-mouse-y                  pic s9(4) comp-5.
         07 ds-mouse-state              pic 9(2)  comp-x.
         07 ds-mouse-moved-flag         pic 9(2)  comp-x.
         07 ds-mouse-over               pic 9(2)  comp-x.
         07 filler                      pic x(3).
78 ds-eb-size         value next - ds-eb-start.

      05 ds-keyboard-event-data
                            redefines ds-mouse-event-data.
         07 ds-char-1.
            09 ds-byte-1                pic 9(2)  comp-x.
         07 ds-char-2.
            09 ds-byte-2                pic 9(2)  comp-x.
         07 filler                      pic x(8).
      05 ds-window-event-data redefines ds-mouse-event-data.
         07 ds-window-x                 pic s9(9) comp-5.
         07 ds-window-y                 pic s9(9) comp-5.
         07 ds-window-command           pic 9(2)  comp-x.
         07 filler                      pic x.
```

The fields in **dssysinf.cpy** of particular interest are `ds-descendant` and `ds-ancestor`. When Dialog System creates an object it gets back a unique identifier for that object called a handle. If an event occurs on that object, Panels V2 passes back to Dialog System the handle of the object as well as the handle of the parent window of the object. These two handles, `ds-descendant` and `ds-ancestor`, are filled whenever an event occurs.

Note: The format of **dssysinf.cpy** has changed from Dialog System V2.1. If you have any Dialog System V2.1 programs that used **dssysinf.cpy**, you must re-compile them.

13.4 Building a Dialog System/Panels V2 Application

The steps you must follow in a Dialog System/Panels V2 application are:

- Establish cooperation between Dialog System and Panels V2.
- Identify the Dialog System objects to Panels V2.
- Perform Panels V2 functions.

To illustrate these steps, the following sections present a simple example of a Panels V2 function that renames a push button. Although a dialog function is available to do this (SET-OBJECT-LABEL), this example illustrates the main features of the Panels V2 call interface.

13.4.1 Establishing Dialog System and Panels V2 Communication

In your COBOL program, initialize Dialog System with a statement like:

```
call "dsgrun" using ds-control-block
                    data-block
                    ds-event-block
```

One of the items returned by Dialog System is `ds-session-id` (this item is stored in the Control Block). `ds-session-id` is the thread identifier of the Dialog System session. For your application to cooperate with Dialog System in its communication with Panels V2, you must make this identifier known to Panels V2. Do this with a statement like:

```
move ds-session-id to p2-mf-reserved
```

where `p2-mf-reserved` is a data item in the Panels V2 parameter block.

This statement establishes the necessary communication link between Dialog System and Panels V2. Now you can make direct Panels V2 calls.

13.4.2 Identifying Dialog System Objects to Panels V2

The MOVE-OBJECT-HANDLE function lets you save the handle of an object in a numeric data item in the Data Block. This ensures the same objects are being referenced by both the Panels V2 and the Dialog System parts of your application. As an example:

```
MOVE-OBJECT-HANDLE RENAME-PB PB-HAND
MOVE-OBJECT-HANDLE RENAME-WIN WIND-HAND
```

stores the handle of a push button named `RENAME-PB` in a numeric item `PB-HAND` and the handle of the window named `RENAME-WIN` in `WIND-HAND`. `RENAME-WIN` and `WIND-HAND` are defined in data definition as:

```
RENAME-HAND             C     4.00
WIND-HAND               C     4.00
```

Now your application (and hence Panels V2), has access to your Dialog System objects.

13.4.3 Perform Panels V2 functions

This fragment shows the code necessary to change the title of a push button.

```
 1 rename-button section.
 2
 3    initialize p2-parameter-block
 4    initialize p2g-button-record
 5    move 250                  to p2g-button-text-length
 6    move pb-hand              to p2-descendant
 7    move ds-session-id        to p2-mf-reserved
 8    move pf-get-button-details to p2-function
 9    call "PANELS2" using p2-parameter-block
10                         p2g-button-record
11                         text-buffer
12    end-call
13    perform varying ndx1 from 30 by -1 until ndx1 = 1 or
14               rename-text(ndx1:1) not = " "
15        continue
16    end-perform
17    move ndx1                 to p2g-button-text-length
18    move pf-set-button-details to p2-function
19    call "PANELS2" using p2-parameter-block
20                         p2g-button-record
21                         rename-text
22    end-call.
```

Line 1:

```
rename-button section.
```

This section illustrates the Panels V2 call interface.

Lines 3-5:

```
    initialize p2-parameter-block
    initialize p2g-button-record
    move 250                  to p2g-button-text-length
```

Initialize some Panels V2 parameters.

Line 6:

```
    move pb-hand              to p2-descendant
```

pb-hand is the handle of the push button. This field is in the Data Block.

Line 7:

```
move ds-session-id         to p2-mf-reserved
```

ds-session-id is the thread identifier of the Dialog System session. See the section *Establishing Dialog System and Panels V2 Communication*.

Line 8:

```
move pf-get-button-details to p2-function
```

pf-get-button-details is a Panels V2 function to retrieve the details of a push button. Because we are only interested in changing the title, all the remaining features are the same.

Lines 9-12:

```
call "PANELS2" using p2-parameter-block
                             p2g-button-record
                             text-buffer
end-call
```

Panels V2 call statement.

Lines 13-16:

```
perform varying ndx1 from 30 by -1 until ndx1 = 1 or
                   rename-text(ndx1:1) not = " "
    continue
end-perform
```

This perform clause removes trailing spaces from rename-text, which contains the new name for the button.

Line 17:

```
move ndx1                  to p2g-button-text-length
```

After removing trailing spaces, ndx1 contains the actual length of the name. p2g-button-text-length is the Panels V2 parameter that contains the length of the title buffer.

Line 18:

```
move pf-set-button-details to p2-function
```

This Panels V2 function changes the attributes of a push button.

Lines 19-22:

```
     call "PANELS2" using p2-parameter-block
                          p2g-button-record
                           rename-text
     end-call.
```

Another Panels V2 call.

Warning: Do not create or delete any objects with Panels V2. Dialog System assumes it is in complete control. For example, if Panels V2 creates an object, the handle is never passed to Dialog System. As a result, any events related to that object by-pass Dialog System.

13.5 Sample Program

Two more extensive applications illustrating the Dialog System/Panels V2 interface are available in the demo directory. One, **Clip**, shows how you can incorporate some basic Panels V2 clipboard read and write functions into your Dialog System application. The other, **Dsp2demo**, shows some clipboard functions, setting colors, and window scrolling.

13.6 Panels V2 User Events

You can generate and receive Panels V2 user events in Dialog System. See the function descriptions for POST-USER-EVENT and GET-USER-EVENT-DATA in the Help. The first 32,000 events are reserved for Micro Focus use.

14 Using the Client/Server Binding

This chapter shows you how the Client/Server Binding works and then describes how to connect your user programs to the generic client/server modules.

14.1 Introduction

The Client/Server Binding enables you to implement a client/server architecture with a Dialog System front end using the Micro Focus Common Communications Interface component (CCI) "under the covers". The Client/Server Binding removes the requirement for you to include communications code in your application by providing two modules called **mfclient** and **mfserver**. These are generic modules which can be used to drive any application.

Based on information contained in a configuration file, the modules manage the communication and transfer of data between themselves and are able to interact with user-defined programs at each end of the link to process this data. **mfclient** is called by a user client program which handles the user interface, and **mfserver** calls a user server program which handles data access and business logic. However, as these user client and server programs are user-defined, they can do anything you require.

The user programs must use the copyfile **mfclisrv.cpy**, described in the section *The mfclisrv.cpy Copyfile*, to interact with the generic client/server modules.

A sample two-tier application, based on the Dialog System CUSTOMER demonstration, is available in the directory **DialogSystem\demo\csbind** of your installation directory. This sample application demonstrates how to use use the Client/Server Binding and is referred to throughout this chapter. For more information on the sample application please see the file *csbind.txt* installed in the **DialogSystem\demo\csbind** directory.

14.2 How the Client/Server Binding Works

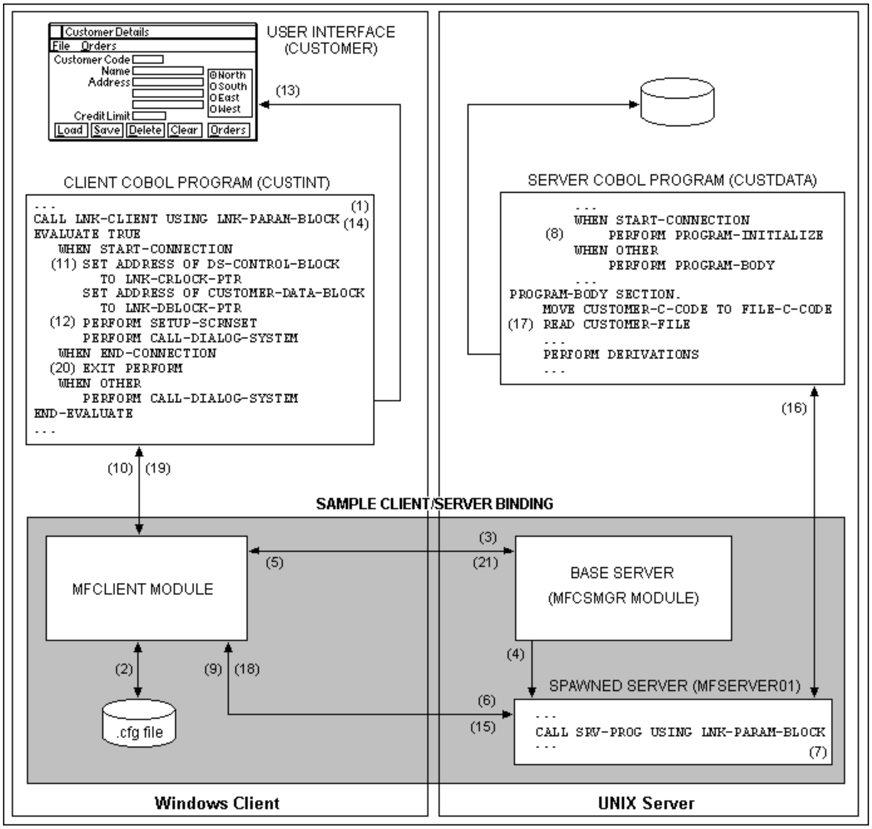
This section illustrates how a two-tier application, created using the Client/Server Binding works.

The Client/Server Binding works by having a non-dedicated copy of **mfserver** running on the server tier; that is, **mfserver** communicates with any and all clients using an agreed server name. This **mfserver** module can be run with its built-in defaults since its only function is to receive the starting and ending communications from the client.

This process is illustrated in the next diagram, Figure 14-1. The flow of information illustrated in the diagram is explained below:

1. The user's client program (**custint**) CALLs **mfclient**.
2. The first time **mfclient** is called, it reads configuration information from the **.cfg** file.
3. **mfserver** receives the connection request.
4. **mfserver** spawns a secondary server for each client. The server name is internally generated, based on the name of the initial server with a numeric ID added (for example, **mfserver01**). Alternatively, you can specify a name in the configuration file.
5. **mfserver** sends the secondary server name (**mfserver01**) back to **mfclient** and terminates the conversation.
6. **mfclient** connects with the secondary server (**mfserver01**), passing parameters obtained from the configuration file via the LNK-PARAM-BLOCK. See the section *The Client/Server Binding Configuration File*.
7. **mfserver01** CALLs the user server program (**custdata**), passing the parameters received from **mfclient** via the Linkage Section.
8. The first time the user server program (**custdata**) is called, it performs any application initialization code and exits the program. Control is returned to **mfserver01**.

Figure 14-1. Client/Server Binding



```
...                                          (1)
CALL LNK-CLIENT USING LNK-PARAN-BLOCK       (14)
EVALUATE TRUE
   WHEN START-CONNECTION
  (11) SET ADDRESS OF DS-CONTROL-BLOCK
          TO LNK-CRLOCK-PTR
       SET ADDRESS OF CUSTOMER-DATA-BLOCK
          TO LNK-DBLOCK-PTR
  (12) PERFORM SETUP-SCRNSET
       PERFORM CALL-DIALOG-SYSTEM
   WHEN END-CONNECTION
  (20) EXIT PERFORM
   WHEN OTHER
       PERFORM CALL-DIALOG-SYSTEM
END-EVALUATE
...
```

```
          ...
          WHEN START-CONNECTION
     (8)      PERFORM PROGRAM-INITIALIZE
          WHEN OTHER
              PERFORM PROGRAM-BODY
          ...
PROGRAM-BODY SECTION.
     MOVE CUSTOMER-C-CODE TO FILE-C-CODE
(17) READ CUSTOMER-FILE
     ...
     PERFORM DERIVATIONS
     ...
```

```
...
CALL SRV-PROG USING LNK-PARAM-BLOCK
...                                  (7)
```

9 **mfserver01** returns control to **mfclient**.

10 **mfclient** ensures contact has been established and returns control to the user client program (**custint**).

11 In the user client program (**custint**), data associated with the user interface is mapped to areas assigned by **mfclient** within the Linkage Section.

12 The user client program (**custint**) calls the user interface. up the screenset and CALLs Dialog System.

13 The user enters input in the user interface (**CUSTOMER**).

14 The user client program CALLs **mfclient** again to pass back the user-entered data (for example a Customer Code) and any other information required by the application server program (**custdata**) via the Linkage Section.

15 **mfclient** passes the data to **mfserver01** via its internal buffer.

16 **mfserver01** CALLs the user server program (**custdata**).

17 The user server program (**custdata**) performs any appropriate data access and business logic and returns the results to **mfserver01** via the Linkage Section.

18 **mfserver01** returns control to **mfclient**.

19 **mfclient** passes the data back to the user client program (**custint**) via the Linkage Section.

20 The user client program (**custint**) CALLs the user interface to display data and accept any new user input.

Steps 14 thru 20 are repeated until the user exits the application.

21 **mfclient** informs the base server (**mfserver**), that the secondary server (**mfserver01**) has terminated.

This use of initial and secondary servers resolves several issues:

- There is no need to have specific servers for different applications. The secondary server is provided with relevant details by the client from the client configuration file. Users can still set up multiple servers, but there is no longer a requirement to do so.

- There are no data access conflicts, as each user has his or her own run unit which ensures that files are always in the same state as when last accessed.

- In the event that a user makes a time-intensive request, only that user will experience any delay. All other users will continue to receive optimum responses. Processing time-intensive requests can cause the client to lock up the interface. You can deal with this in the following way:

You can issue an asynchronous request which returns a request id to the program. You then make regular checks for the completion of the request using the id which was returned. An example of using asynchronous requests is given in the section *Connecting Your Client Program to mfclient*.

14.3 Connecting Your Programs to the Generic Modules

The following sections show you how to:

- Connect your user programs to the generic client and server modules.
- Prepare the communications link.

14.3.1 Connecting Your Client Application to mfclient

This section shows you how to connect your client program to the **mfclient** module, which handles communications with the server. The **mfclient** and **mfserver** modules pass information to each other via a parameter block described in the **mfclisrv.cpy** copyfile.

The modules also use the same parameter block to pass information to any user programs they have been requested to call in the configuration files. For details on the LNK-PARAM-BLOCK, see the section *Connecting Your Server Application to mfserver*.

To connect your client program calling the user interface, you must add code to pass parameters to **mfclient**.

To use the Client/Server Binding, your client program will need to include code similar to that below. This is an extract of the user interface program (**custint.cbl**) supplied with Dialog System to demonstrate how to use the Client/Server Binding.

```
$SET ANS85
 WORKING-STORAGE SECTION.
 COPY "MFCLISRV.CPY".
 LINKAGE SECTION.
* The following two copyfiles are used by Dialog System
* user interfaces to communicate with the COBOL program
* which calls the interface
 COPY "DS-CNTRL.V1".
 COPY "CUSTOMER.CPB".
 PROCEDURE DIVISION.
 CLIENT-CONTROL SECTION.
     PERFORM UNTIL END-CONNECTION
         CALL LNK-CLIENT USING LNK-PARAM-BLOCK
         EVALUATE TRUE
          WHEN START-CONNECTION
             SET ADDRESS OF DS-CONTROL-BLOCK
                                          TO LNK-CBLOCK-PTR
             SET ADDRESS OF CUSTOMER-DATA-BLOCK
                                          TO LNK-DBLOCK-PTR
             PERFORM SETUP-SCRNSET
             PERFORM CALL-DIALOG-SYSTEM
          WHEN END-CONNECTION
             EXIT PERFORM
          WHEN OTHER
             PERFORM CALL-DIALOG-SYSTEM
         END-EVALUATE
         IF CUSTOMER-EXIT-FLG-TRUE
            SET CLIENT-ENDING TO TRUE
         END-IF
     END-PERFORM.
 CLIENT-CONTROL-END.
     STOP RUN.
```

You can add code to enable client counting, handle error message displays yourself or handle asynchronous requests. For details, see the sections *Connecting Your Client Program to mfclient* and *Connecting Your Server Program to mfserver*.

14.3.2 Connecting Your Server Application to mfserver

This section shows you how to connect your server program, which performs data access and/or business logic, to the **mfserver** module which handles communications with the client. The **mfclient** and

mfserver modules pass information to each other via a parameter block described in the **mfclisrv.cpy** copyfile. The modules also use the same parameter block to pass information to any user programs they have been requested to call in the configuration files.

The following information does not apply if you are using an existing application as your server program. For details on using the Client/Server Binding in that way, see the section *Running a Client/Server Binding Application*.

To use the Client/Server Binding:

- Your server program will need to include in the Linkage Section the binding copyfile, **mfclisrv.cpy**, and any copyfiles required to pass information between your client program and the user interface.
- Modify the Procedure Division header to include "LNK-PARAM-BLOCK", which passes parameters from **mfserver**.
- Associate addresses passed by **mfserver** in the **mfclisrv.cpy** with the copyfiles used by your user interface, in this example DS-CONTROL-BLOCK and the SCREENSET-DATA-BLOCK are the data structures defined in **ds-cntrl.cpy** and **customer.cpb**. For details on the LNK-PARAM-BLOCK, see the section *Connecting Your Server Program to mfserver*.

This is shown below in the code extract of the server program (**custdata.cbl**) used to run the Client/Server Binding.

```
 LINKAGE SECTION.
 COPY "DS-CNTRL.V1".
 COPY "CUSTOMER.CPB".
 COPY "MFCLISRV.CPY".
 PROCEDURE DIVISION USING LNK-PARAM-BLOCK.
 CONTROLLING SECTION.
*-----------------------------------------------------------*
*  ASSOCIATE THE DIALOG SYSTEM COPYBOOKS WITH AREAS RESERVED
*  FOR THEM WITHIN LNK-PARAM-BLOCK
*-----------------------------------------------------------*
     SET ADDRESS OF DS-CONTROL-BLOCK TO LNK-CBLOCK-PTR.
     SET ADDRESS OF CUSTOMER-DATA-BLOCK TO LNK-DBLOCK-PTR.
     EVALUATE TRUE
      WHEN START-CONNECTION
         PERFORM PROGRAM-INITIALIZE
      WHEN OTHER
         PERFORM PROGRAM-BODY
     END-EVALUATE.
     EXIT PROGRAM.
 PROGRAM-INITIALIZE SECTION.
     OPEN I-O CUSTOMER-FILE.
 PROGRAM-BODY SECTION.
     MOVE CUSTOMER-C-CODE TO FILE-C-CODE
     READ CUSTOMER-FILE
     ..........
     PERFORM DERIVATIONS
```

You can add code to handle error message displays yourself. For details, see the section *Connecting Your Server Program to mfserver*.

14.3.3 Preparing a Communications Link

The Client/Server Binding controls the communications between your client and server programs based on the contents of the configuration file. That means there is no complicated communications programming for you to do.

To run the demonstration, use the appropriate configuration (**.cfg**) file unmodified. For your own application, you must copy and modify the contents of one of these configuration files. The configuration file is an ASCII text file containing parameters allowing you to specify key information such as:

- The name of the user program to be called by mfserver.
- The maximum number of clients allowed to connect to the server.

For more details on using configuration files with the Client/Server Binding, see the section *The Client/Server Binding Configuration File*.

Notes:

- This document assumes your communications link using CCI is already correctly installed, configured and working on both your client and server machines.
- You must use a network software vendor supported by Micro Focus CCI. For full details on supported vendor's software, see the product description on the Micro Focus World-wide Web homepage, contact

your Sales representative or see the documentation supplied with the Micro Focus software.

14.4 Before Using the Client/Server Binding

You can use the client/server binding with an existing standalone application or an application architectured for client/server. For more information on using existing standalone applications, see the section *Running a Client/Server Binding Application*.

In either case, your application contains the following (examples of the supplied Dialog System demonstration programs are shown):

	Two-tier Demo Program	**Standalone Demo Program**
User Interface	**CUSTOMER.gs** (GUI)	
COBOL code to call the interface	**custint.cbl**	**usexsrv.cbl**
COBOL code to perform data access and apply business logic	**custdata.cbl**	**customer.cbl**

You can use the client/server binding instead of creating your own middleware code. For this, you need to:

- Create a configuration(**.cfg**) file to control the behavior of the **mfclient** and **mfserver** modules and the communications connection.

  The configuration file is an ASCII text file containing parameters allowing you to specify key information such as:

  - Which communications protocol will be used.
  - The name of the user program to be called by mfserver.

  This configuration file format is described in detail in the section *The Client/Server Binding Configuration File*. Sample configuration files (**custgui.cfg** and **customer.cfg**) are provided with the client/server binding demonstration programs.

You can have separate configuration files for each application or you can supply multiple entries in a single file. If using multiple entries, each one must have its own tag name followed by the list of parameters to which the tag relates. You also need to supply the tag name to be used, by putting the relevant name in `lnk-tagname` in your client interface program. This technique can be used to reduce your command line length by specifying your multiple entries in a file called **mfclisrv.cfg** for which the binding will search if no other name is specified. This technique can also be used to allow you to drive different applications from a single menu.

- Add code to your client program to call the **mfclient** module with linkage parameters. This is explained in the section *Connecting Your Client Application to mfclient*.
- Add code to your server program to enable it to be called with linkage parameters from the **mfserver** module. This is explained in the section *Connecting Your Server Application to mfserver*.

14.4.1 The mfclisrv.cpy Copyfile

The **mfclient** and **mfserver** modules pass information to each other via the parameter block defined in the copyfile **mfclisrv.cpy**. For example, the information includes addresses for the Dialog System `screensetdata-block` and `ds-control-block`. The modules also use this parameter block to pass information to any user programs they have been requested to call in the configuration files. The **mfclisrv.cpy** copyfile must be included in any of your COBOL programs which use the **mfclient** and **mfserver** modules.

The **mfclisrv.cpy** copyfile is installed in the SOURCE directory of your Net Express base installation directory.

14.4.2 The Client/Server Binding Configuration File

This section describes the configuration file for the client/server binding. This configuration file controls the behavior of the **mfclient** and **mfserver** modules as well as the communications link.

You create one or more configuration files for your application. You can have as many configuration files as your application requires, but each file must have a **.cfg** extension. If you do not supply a configuration filename yourself, the program will default to using the name **mfclisrv.cfg**. You do not need a configuration file, but if you do not supply one, the demonstration programs supplied with the client/server binding are run by default.

If your client/server application is incorrectly set up, any invalid entries found in the configuration file cause the programs reading them to terminate with a message displayed on the screen detailing the invalid entries.

These errors are also logged in the **mfclisrv.log** file, which is created in the current directory, or in the directory indicated by the MFLOGDIR environment variable. This enables a server which is not directly connected to a terminal to log messages regarding problems it encounters.

14.4.2.1 Possible Entries for the Configuration File

Below are possible entries for the **mfclisrv.cfg** configuration file. Entries are listed in alphabetical order. Any entries which you do not specify in your configuration file will assume the default values.

```
**********************************************************
* Micro Focus - Client/Server Module Configuration File
**********************************************************
```

[mf-clisrv]

cblksize=*nnnn* PIC X(4) COMP-X [default:0]

Size of Dialog System control block.

clierrprog=*xxxxxx* PIC X(128) [default:none]

Name of program to handle communication errors instead of mfclient. The name 'SAME' will use the calling program to handle any mfclient errors.

commsapi=*xxxx*

PIC X(4) [default:CCI]

API used for communications (CCI/NONE). The special entry ’NONE’ can be used when developing two tier applications to allow testing to be undertaken on a single PC without any communications products.

Data is passed directly between **mfclient** and the user server program, bypassing **mfserver** and the communications requirement.

Note: This option cannot be used if an existing application is being deployed as the user server program.

compress=*nnn*

PIC 9(3) [default:000]

Compression routine indicator. The number indicates the name of the compression routine to be used, that is, 001 uses the routine CBLDC001. Zero indicates data compression will not be used.

dblksize=*nnnn*

PIC X(4) COMP-X [default:0]

Size of the user data block. When using the binding modules with Dialog System, this entry must match or be greater than the size of the Data Block generated by Dialog System. If the data fields used by the screenset are changed and the copyfile regenerated, change this entry to reflect the size of the new Data Block. When using multiple screensets, this entry should reflect the size of the largest screenset Data Block.

Note: Unpredictable results can occur if this entry is less than the actual size of the Data Block being used.

eblksize=*nnnn*

PIC X(4) COMP-X [default:0]

Size of optional Dialog System Event block.

machinename=

PIC X(34)

Name of the machine on which the server specified in the servername entry is located. This prevents the system searching for the first match of the defined name.

maxtrans=*xxxx*

PIC 99 [default:0]

Maximum transfer package size in Kilobytes. If the total buffer size exceeds this amount, the system will make multiple transfers of ’maxtrans’ size until the total buffer is transferred. The accepted values are 1-62.

midconfig=*xxxxx*

PIC X(128) [default:none]

Name of the configuration file to be used by **mfclient** when called by **mfserver** as part of a cross-tier solution. Specifying this entry causes mfserver to act as a router passing data to a local mfclient module.

This **mfclient** module uses the configuration file to locate and communicate with another server. This allows protocols and communications API's to be changed across the machines used.

protocol=*xxxxxx*

PIC X(8) [default:CCITC32]

The CCI protocol to be used. This entry is applicable only if commsapi is set to CCI. You can specify any of the supported CCI protocols:

- Novel IPX (enter CCIIX32)
- NetBEUI (enter CCINB32)
- Dynamic Data Exchange (enter CCIDE32)
- TCP/IP (enter CCITC32)

If commsapi is set to CCI (the default setting) and you do not specify a protocol, the client/server binding defaults to TCP/IP (CCITC32).

scrntype=*xxxx*

PIC X(4) [default:none]

Required interface indicator when multiple types are supported, that is, using the character control block to run both character and GUI interfaces. scrntype is not validated by the client/server binding modules

servername=*xxx*

PIC X(14) [default:MFCLISRV]

Servername to be used for communications.

setenv=name value

PIC X(148) [default:none]

Environment variable to be set on the server before the execution of any server programs. The format is:

variable name PIC X(20) variable value PIC X(128)

The name and value fields must be separated by at least one space and up to nine setenv entries can be specified.

srvanim=*x*

PIC X(16) [default:N]

Y or N. Setting this parameter to Y enables animation of the user programs running on the server. This means that **mfserver** does not have to be stopped and restarted with COBSW=+A as this is set up dynamically.

On UNIX systems only, you can specify a value of **x,*filename*** to enable cross-session animation of your programs. See the section *Animating your Application* for more details.

srverrprog=*xxxxxx*

PIC X(128) [default:none]

Name of program to handle communication errors instead of mfserver. The name 'SAME' will use the name set in the srvprog entry for **mfserver** errors.

srvprog=*xxxxxx*

PIC X(128) [default:custdata]

Name of user program to be called by **mfserver**.

srvtier=*xxxxxx*

PIC X(128) [default:mfserver]

This holds the name of the server tier program. When specified, it indicates that **mfserver** will be called by this program.

subserver=*xxxx*

PIC X(14) [default:none]

Basename of the server to be used for communications after the first contact with the main server instead of a name based on the initial servername. For example, the default servername is MFCLISRV and subservers will be named MFCLISRV00001, MFCLISRV00002..., but setting and entry of say NEWSERV here would result in subservers named NEWSERV00001, NEWSERV00002. This allows application specific subserver names to be used without the need to change the base server name.

timeout=*nnnnn*

PIC X(4) COMP-5 [default: 120 secs)

Use to override the system timeout default. The timeout period is specified in 1/10 (tenths) of a second. Previously issued calls retain the timeout period that was in use when they were originally invoked.

ublksize=*nnnn*

PIC X(4) COMP-X [default:0]

Size of optional user Data Block.

useraudit=*x*

PIC X [default:N]

Toggle to control the logging of client connect and disconnect details. When set, the following details are logged by both client and server:

Date, Time, connect/disconnect indicator, Servername, Machinename, and Protocol.

14.4.2.2 Minimum Required Configuration File Entries

The minimum number of entries required in the configuration files varies depending on which of the following factors are being used:

- The application (for example Dialog System).
- The version of Dialog System (hence the size of the Dialog System control block required).
- The functions of Dialog System (for example callouts).
- The number of servers.
- The communications protocol (for example CCITCP).

The following table shows the different entries needed for Dialog applications which are and are not using an existing Dialog System program as the server.

	Not using a DS program as the server	**Using a DS program as the server**
dblksize	The size of the Data Block.	The size of the Data Block.
srvprog	The name of the program that **mfserver** calls to perform the server-side processing. This is the user program on the server machine (that is, the COBOL program performing data access and business logic).	

	Not using a DS program as the server	**Using a DS program as the server**
srvtier		The name of the program that the base server will spawn. By default, this is **mfserver**, which should be changed only if you wish to use an existing Dialog System program as the server. This existing program also contains the COBOL code to perform data access and business logic.
Cblksize	The size of the Dialog System control block (if you are not using the **ds-cntrl.r1** copybook).	The size of the Dialog System control block (if you are not using the **ds-cntr.r1** copybook).
Protocol		The protocol to use. For example, CCITC32 for TCP.

When running multiple servers, you must supply the name of the server to be used via the servername entry.

Any configuration file entries which are unset assume the specified default values.

If you are not using TCP/IP, you must set the protocol entry.

14.4.2.3 Locating The Configuration File

You can specify where to locate the configuration file. The **mfclient** and **mfserver** programs attempt to locate the configuration file in the following ways:

1. They look for the MFCSCFG environment variable and use the supplied filename if one is specified.
2. They examine the command line and use any filename defined there in preference to that supplied via the MFCSCFG environment variable.
3. If MFCSCFG is not set or contains no value, or if no filename has been specified on the command line, the default name **mfclisrv.cfg** is assumed and searched for in the current directory.

4 If **mfclisrv.cfg** is not found, the default settings for the configuration entries described earlier in this document will be used.

In each case a full path of up to 128 characters can be used to identify the location of the configuration file.

14.5 Connecting Your Client Program to mfclient

In order to create programs using the client/server binding, add code similar to that described below to your client and server programs, so they can control the connection correctly. The amount of code required is quite small. Comments have been included to help you understand what the code is doing.

```
 WORKING-STORAGE SECTION.
 COPY "mfclisrv.cpy".

 LINKAGE SECTION.

 COPY "DS-CNTRL.V1".
 COPY "CUSTOMER.CPB".

 PROCEDURE DIVISION.
 Client-Control SECTION.

*  The main loop is repeated until the connection with
*  the server ends

     PERFORM UNTIL End-Connection

* 'lnk-client' holds the name 'mfclient'
*  The first time through we initialize the system and
*  establish contact with the server.

         CALL lnk-client USING lnk-param-block

         EVALUATE TRUE
          WHEN start-connection

*  Make the DS control block and the customer Data Block
*  accessible by assigning them address allocated by
*  ’mfclient’.

             SET ADDRESS OF ds-control-block TO lnkcblock-ptr
             SET ADDRESS OF customer-data-block
                                             TO lnkdblock-ptr

*  Having successfully established contact with the server,
*  we complete local initialization and make the first call
*  to Dialog System.

             PERFORM Setup-Scrnset
             PERFORM Call-Dialog-System
          WHEN end-connection
             EXIT PERFORM
          WHEN OTHER
             PERFORM Call-Dialog-System
         END-EVALUATE

*  Check if the exit flag has been set by the user on the
*  screen

         IF customer-exit-flg-true
             SET client-ending TO TRUE
         END-IF
     END-PERFORM
     STOP RUN.
```

Note: The client program has the Dialog System copybook in linkage and maps it to an address provided by the binding. This copybook has three other 01-level items which are not mapped. These are the Dialog System version number and the Data Block version number. The reason they are not mapped is that their values would not be picked up by mapping, and it is the values they contain that we need. You can hard code the Dialog System version number in the program, as this very rarely changes. The Data Block version number, however, does change, and putting a fixed value in your program means that you have to remember to update it whenever the screenset is modified. To make this easier, a support program used by the binding can extract the screenset version number for you and allow you to set it dynamically. To use this program, add the following items to working storage:

```
    01  ws-null              PIC X(4) COMP-X.
    01  ws-scrnset-ver       PIC X(4) COMP-X.
    01  ws-ret-stat          PIC X COMP-X VALUE 0.
```

Add the following to the Procedure Division where the screenset details are being set up. The screenset name being used must include the correct extension. You would set `ds-version-no` to 3 for extensions of **.rs**.

```
     MOVE "CUSTOMER.gs" TO ds-set-name
     MOVE 2 TO ds-version-no
     CALL "dsdblksz" USING
                         ds-set-name
                         ws-null
                         ws-scrnset-ver
                         ws-ret-stat
     END-CALL
     MOVE ws-scrnset-ver to ds-data-block-version-no
```

If you want to control the number of clients running an application, or you choose to handle error message displays yourself, add code similar to the following to your program’s initial EVALUATE statement.

```
      WHEN TOO-MANY-CLIENTS
         PERFORM OVER-CLIENT-LIMIT
      WHEN COMMS-ERROR
         PERFORM SHOW-ERROR
...

 OVER-CLIENT-LIMIT SECTION.
     DISPLAY SPACES AT 0101 WITH BACKGROUND-COLOR 7
       "MAXIMUM NUMBER OF CLIENTS EXCEEDED - SESSION ENDED"
        AT 1012 WITH FOREGROUND-COLOR 4
     SET CUSTOMER-EXIT-FLG-TRUE
         CLIENT-ENDING TO TRUE
      EXIT.

 SHOW-ERROR SECTION.
     DISPLAY LNK-ERROR-LOC AT 2201
         LNK-ERROR-MSG AT 2301 WITH SIZE LNK-ERROR-MSG-LEN.
 SHOW-ERROR-EXIT.
     EXIT.
```

If you want to handle asynchronous requests, add additional code similar to the following to the EVALUATE statement:

```
      WHEN START-CONNECTION
         PERFORM GET-USER-INPUT
         IF MAKE-ASYNC-REQUEST  <* USER ASYNCHRONOUS OPTION
             SET ASYNC-REQUEST TO TRUE
         END-IF
      WHEN ASYNC-OK
         SET TEST-ASYNC-RESULT TO TRUE
         PERFORM DELAY-LOOP
      WHEN ASYNC-INCOMPLETE
         DISPLAY "REQUEST STILL BEING PROCESSED " AT 1010
         PERFORM DELAY-LOOP
         SET TEST-ASYNC-RESULT TO TRUE
      WHEN RESULT-OK
         DISPLAY "REQUEST COMPLETED             " AT 1010
         PERFORM GET-USER-INPUT
      WHEN ASYNC-NOT-STARTED
      WHEN ASYNC-FAILED
         DISPLAY "ASYNCHRONOUS REQUEST FAILURE  " AT 1010
         PERFORM SHOW-ERROR
         PERFORM GET-USER-INPUT
      WHEN COMMS-ERROR
         PERFORM SHOW-ERROR
```

14.6 Connecting your Server Program to mfserver

```
 WORKING-STORAGE SECTION.

 LINKAGE SECTION.

     COPY "DS-CNTRL.V1".
     COPY "CUSTOMER.CPB".
     COPY "mfclisrv.cpy".

 PROCEDURE DIVISION USING lnk-param-block.
 Controlling SECTION.
*-----------------------------------------------------------*
*  Associate the Dialog System copy books with their
*  location within the CS binding parameter block
*-----------------------------------------------------------*

     SET ADDRESS OF ds-control-block TO lnk-cblock-ptr.
     SET ADDRESS OF customer-data-block TO lnk-dblock-ptr.
     EVALUATE TRUE
      WHEN start-connection
         PERFORM Program-Initialize
      WHEN customer-exit-flg-true
         PERFORM Program-Terminate
      WHEN OTHER
         PERFORM Program-Body
```

```
    END-EVALUATE.
    EXIT PROGRAM.
```

If you choose to handle error message displays yourself, add code similar to the following to your program's initial EVALUATE statement.

```
     WHEN COMMS-ERROR
        PERFORM SHOW-ERROR
. . .
SHOW-ERROR SECTION.
    DISPLAY LNK-ERROR-LOC AT 2201
            LNK-ERROR-MSG AT 2301
                          WITH SIZE LNK-ERROR-MSG-LEN.
SHOW-ERROR-EXIT.
    EXIT.
```

14.7 Running a Client/Server Binding Application

The **mfserver** module is provided in **.int** code format so that it can be run as it is, generated into **.gnt** code, or linked with other programs to create executable modules (UNIX only).

The **mfclient** and **mfserver** modules are run as standard COBOL programs, with both the client and server components started manually.

When running an existing stand-alone program as a server, specify its name, in the **srvtier** configuration entry, as the program which the base server should run (**srvtier**). Communications are handled by a copy of the **mfserver** module renamed **dsgrun** created as follows:

- For non-UNIX servers, you must copy the **mfserver.int** file to **DSGRUN.int**.
- On UNIX, you can link the **mfserver.int** file to **DSGRUN.int** instead of creating a separate program. On UNIX, you must also create a cobconfig file containing the following entry:

  ```
  set program_search_order=3
  ```

and set the COBCONFIG environment variable to point to this file. This ensures that the call to DSGRUN in the standalone application will call the renamed mfserver module.

For details on the cobconfig file and environment variable, see your UNIX COBOL documentation. For details on linking and building executable modules, see the documentation for your chosen COBOL development environment.

The main benefit of using an existing program as the server is that you can have an application which runs on a single PC or a PC network, and can be deployed as a client/server application simply by using the binding and providing a single user interface program.

To run the client/server binding, first start the server program and then start the client program.

The command line to start the server program on UNIX is:

```
cobrun mfserver [-b] [-p protocol] [-s server-name] [-v]
```

and on Windows and OS/2:

```
runw mfserver [-p protocol] [-s server-name] [-v]
```

where:

-b (UNIX only) allows you to run the server as a background process. The command line should be terminated with the ampersand (&) character. Any error messages produced by the server are stored in the log file, **mfclisrv.log**.

-p *protocol* specifies the communications protocol.

-s *server-name* indicates a server-name other than the default server-name MFCLISRV.

-v displays the mfclient version number.

The command line to start the client program is:

```
runw program-name [config-filename]
```

where:

program-name Specifies the name of the user interface program.

config-filename Specifies the name of the configuration file to be used.

The client searches for its configuration file, using the name specified by the MFCSCFG environment variable, then using the name specified on the command line. If neither exists, it searches for a configuration file called **mfclisrv.cfg** (in this case, you must have created a file of this name).

For further details on these and other configuration file entries or full details on locating the configuration file, see the section *The Client/Server Binding Configuration File*.

For full details on running COBOL programs, see your COBOL system documentation.

14.8 Animating Your Application

You can animate your client program using the standard animation facilities of Net Express.

You can animate your server program by setting a configuration file parameter, **srvanim=y** (see the section *Possible Entries for the Configuration File*).

On PC servers, the Animator is started automatically when a client connects to the server if **srvanim=y** is set.

On UNIX servers, the Animator is run on the terminal from which **mfserver** was started, and so requires **mfserver** to be running in foreground mode. This can cause problems as it is preferable to run **mfserver** as a background process. A further restriction is that only one user can animate through **mfserver** at any one time. These problems can be avoided by setting **srvanim=x,*filename*** where *filename* is a file created using the **touch** command. On the terminal that you want to use to display Animator output, set **COBANIM_2=animator** and then run the command:

```
anim filename
```

On the terminal that you want to use for your standard input and output, run your application in the normal way, having added **srvanim=x,*filename*** to your configuration file.

14.9 Managing the Server

The behavior of the server can be altered by running the program **mfcsmgr** and passing it the required parameters and values. These parameters can be divided into two groups:

- Location

  The parameters in this group allow you to locate the target server, and can have settings m, p, and s.

- Action.

  These parameters control how the target server will be affected. They can have settings a, c, o, r and t. Parameters o, r and t must be specified singly as they generate an error message if combined with any other parameters.

14.9.1 Shutting Down mfserver

The spawned servers are terminated by the client program when the client makes a normal exit.

You must terminate the initial server program manually as it continues to run even if it has no clients.

14.9.2 Managing Authorization Passwords

To prevent accidental shutdown of an active server, it is possible to assign a password to each server. This password must be supplied before the server can be stopped or its parameters changed.

14.9.3 Setting the Maximum Number of Clients

The default number of clients the server can support is 65535, but you can set a lower limit using one of the mfcsmgr functions. The number you select is stored in the password file and will be used each time the server is started.

14.9.4 Enabling Server Override

The server can support override options for the server-name, protocol and machine-name. As the server does not use a configuration file, the override parameters are supplied using the **mfcsmgr** program.

This generates a short exchange between the client and the server which processes the selected function.

The **mfcsmgr** command line syntax for this is:

On Windows or OS/2 enter:

```
run mfcsmgr [-a] [-c nnnnn] [-d] [-i filename]
           [-m machine-name] [-p protocol]
           [-s server-name] [-o m,machinename]
           [-o p,protocol] [-o r]
           [-o s,servername] [-t] [-v]
```

or on UNIX enter:

```
cobrun mfcsmgr [-a] [-c nnnnn] [-d]
           [-i filename] [-m machine-name]
           [-p protocol] [-s server-name]
           [-o m,machinename] [-o p,protocol]
           [-o r] [-o s,servername] [-t] [-v]
```

where:

-a	Changes the authorization password associated with the target server.
-c *nnnnn*	Sets the number of clients the server can support.
-d	Deletes local override file when client exists.

-i *filename*	Installs the specified file in the client's local directory when the client exists.
-m *machine-name*	Specifies the machine-name on which the target server is running. This is required if the same server-name is used on more than one platform.
-p *protocol*	Specifies the communications protocol (for example CCITCP or CCITC32) being used.
-s *server-name*	Indicates a server name other than the default MFCLISRV.
-o m, *machinename*	Specifies the machine-name on which the overriding is running.
-o p, *protocol*	Accesses the overriding server.
-o r	Resets active override. Revert to original settings.
-o s, *servername*	Overrides connections to the target server by directing them to this server.
-t	Terminates the server.
-v	Displays the **mfclient** version number.

All the above flags can be specified with a - or /, and are not case sensitive.

14.10 Advanced Topics

This section covers the following advanced topics:

- Creating audit trails.
- Overriding configuration file entries.
- Using the in-line configuration facility.
- Reduced data transfer facility.
- Server controlled file management facility.

14.10.1 Creating Audit Trails

The client/server binding allows you to create an audit trail of dates and times of clients connecting to servers. To enable this feature, simply set the ’useraudit=y’ entry in your configuration file. This information is created in the system log file described in the section *The System Error/Message Log*.

14.10.2 Overriding Configuration File Entries

In the start-up process for each client, the client/server:

1. Reads the configuration file.
2. Searches the client/server binding for a file called **mfcsovrd.cfg** in the same location as the system log file. If the file is found, its contents will override any parameters previously set up using the configuration file.

   The format of the override file is the same as the standard configuration file, with the addition of the entry **override-cntrl**. This entry is used to indicate the subject of the override and can be:

   - A servername.

     When using a servername, any clients using that server will have their parameters modified by the contents of the override file.

   - A tagname.

     If the entry is set to tagname, only clients using the selected tagname will have their parameters changed.

The client override facility can be used, for example, if a server is unavailable and applications need to be run on another machine. You can change individual configuration files, but a single override file can be used to re-route any number of applications.

The override facility can also be used on the server, but in this case the server machine needs to be up and running. As before, either servernames or tagnames can be overridden.

An entry is added to the log file whenever an override file is detected and all parameters which are subject to the override process are logged.

The following example shows how the override facility can be used to re-route clients to another server:

```
[OVERRIDE-CNTRL]
OVERRIDE=SERVERNAME
[OLDSERVER]
SERVERNAME=NEWSERVER
```

In this example, the **[override-cntrl]** section specifies that the subject of the override is a servername (override=servername) and then, under the old servername (**[oldserver]**), the new servername (**servername=newserver**) is specified. This results in the following log file entries:

```
20/04/1997 11:01:02 Using Local File: mfcsovrd.cfg
Overriding Entries for Servername:OLDSERVER
servername=newserver
20/04/1997 11:01:02 Override Completed:
```

The following example shows how the override facility can be used to override the server being used by all those clients which are using a specified tag:

```
[OVERRIDE-CNTRL]
OVERRIDE=TAGNAME
[MF-CLISRV]
SERVERNAME=NEWSERV
```

In this example, the **[override=cntrl]** section specifies that the subject of the override is a tagname (**override=tagname**) and then, under the tagname (**[mf-clisrv]**), the parameter to be overridden (in this case the servername) is specified (**servername=newserv**). This results in the following log file entries:

```
20/04/1997 11:04:02 Using Local File: mfcsovrd.cfg
Overriding Entries for Tagname:MF-CLISRV
servername=newserver
20/04/1997 11:04:02 Override Completed:
```

14.10.3 Using the In-line Configuration Facility

One of the most powerful features of the client/server binding is the ability to control the communications requirements from configuration files. This does however mean that end users can affect the way an application runs by changing entries in these files. This may not be desirable. It is possible to provide all the parameters to an application in the client program. This means that the end user cannot alter the way an application behaves because it no longer uses a configuration file. Flexibility is maintained in that you still have no need to get into detailed communications code: you simply supply the required parameters within your client program.

The process is started by setting `load-inlinecfg` and completed by setting `end-inline-cfg` in your **mfclisrv.cpy** copyfile in your client program. Each parameter entry is loaded into the `lnkerror-msg` field and **mfclient** is called to process it. The parameters have to be supplied before the main processing loop is entered. See the example below.

```
WORKING-STORAGE SECTION.

01  ws-null          PIC X(4) COMP-X.
01  ws-scrnset-ver   PIC X(4) COMP-X.
01  ws-ret-stat      PIC X COMP-X VALUE 0.

COPY "mfclisrv.cpy".

01  dialog-system    PIC X(48).

LINKAGE SECTION.

COPY "DS-CNTRL.V1".
COPY "CUSTOMER.CPB".

PROCEDURE DIVISION.

Client-Control SECTION.

    SET load-inline-cfg TO TRUE
    MOVE "clierrprog=same" TO lnk-error-msg
    CALL lnk-client USING lnk-param-block

    MOVE "srverrprog=same" TO lnk-error-msg
    CALL lnk-client USING lnk-param-block
```

```
     MOVE "servername=mainserv" TO lnk-error-msg
     CALL lnk-client USING lnk-param-block

     SET end-inline-cfg TO TRUE

*  The main loop is repeated until the connection with
*  the server ends

     PERFORM UNTIL End-Connection

*  'lnk-client' holds the name 'mfclient'
*  The first time through we initialize the system and
*  establish contact with the server.

         CALL lnk-client USING lnk-param-block

         EVALUATE TRUE
          WHEN start-connection

   ........ The rest of the program is standard from this
   ........... point onwards  ............
```

It is also possible to combine an external configuration file with the inline configuration facility. This is ideal because the end-user can modify the system to his own requirements, but the final control remains within the client program. The configuration file is processed first, followed by the inline entries. This means that any unwanted parameters supplied in the configuration file can be overridden by the inline entry. In this case the process is started by setting `use-combined-cfg` and completed as before by setting `end-inline-cfg`.

```
 WORKING-STORAGE SECTION.

 01  ws-null          PIC X(4) COMP-X.
 01  ws-scrnset-ver   PIC X(4) COMP-X.
 01  ws-ret-stat      PIC X COMP-X VALUE 0.

 COPY "mfclisrv.cpy".

 01  dialog-system    PIC X(48).

 LINKAGE SECTION.

 COPY "DS-CNTRL.V1".
 COPY "CUSTOMER.CPB".
 PROCEDURE DIVISION.
 Client-Control SECTION.

     SET use-combined-cfg TO TRUE
     CALL lnk-client USING lnk-param-block
     SET load-inline-cfg TO TRUE
     MOVE "servername=mainserv" TO lnk-error-msg
     CALL lnk-client USING lnk-param-block
     SET end-inline-cfg TO TRUE

*  The main loop is repeated until the connection with
*  the server ends

     PERFORM UNTIL End-Connection

*  'lnk-client' holds the name 'mfclient'
*  The first time through we initialize the system and
*  establish contact with the server.

         CALL lnk-client USING lnk-param-block

         EVALUATE TRUE
          WHEN start-connection

   ....... The rest of the program is standard from this
   .......... point onwards  ............
```

14.10.4 Reduced Data Transfer Facility

Reduced Data Transfer (RDT) provides a way for you to control the amount of data being passed across the network and for you to limit this data to the absolute minimum required to achieve the desired result.

The client/server binding assigns a buffer large enough to hold the Data Blocks defined in the configuration file. This buffer is transferred back and forth whenever control is shifted between the client and server programs. This can impose a load on the network which some users may find unacceptable. Imagine you have a buffer of 24K which contains a 22K record area. If the file which holds these records has a 10 byte key, you can see that sending the key from client to server uses 24K when only 10 bytes are actually required. In reality, you need one or two bytes more than this, but it is much less than the size of the entire buffer.

RDT requires a control flag (use-rdt) and three parameters to be set in **mfclisrv.cpy** in your client program:

lnk-usr-fcode
User function indicator. Lets the user server program know what to do with the data that has just arrived.

Lnk-usr-retcode
Buffer start point. This indicates which of the four data areas will be used as the start point for the transfer:

1 Dialog System control block.

2 Data Block.

3 Dialog System event block.

4 User Data Block.

The numbers 11 through 14 can also be used and indicate the same address areas but use data compression on the transfer. Data compression must have been enabled via a configuration file entry, or the base (uncompressed) option will be used. A value of zero results in nothing being transferred. This allows you to have a NULL operation without changing the flow of your code by adding various IF statements. This is useful if you choose to process certain functions locally in order to further reduce network traffic.

Lnk-data-length
The length of data to be transferred.

For example, consider an application which allows you to ADD, DELETE and LOAD customer details to and from an index file. The record key for the customer details is a customer code. The application also has a CLEAR option to clear all customer information from the interface. The application is the customer example program installed with this product. The code extract below is based on this example application, and we are using the `user-data-block` area to hold the record key which is six bytes long. The Data Block area (dblksize) used by the application to pass the customer record details between the client and the server is called customer-data-block. The data item `customer-c-code` is a 6-byte data item within `customer-data-block` and is used to store the record key.

On the client, the code would be similar to that shown below.

```
EVALUATE TRUE
 WHEN customer-load-flg-true
```

```
*  User has entered customer code and selected the "LOAD"
*  option on the interface to read and display the customer
*  details relating to that code.

            MOVE customer-c-code TO user-data-block
            SET use-rdt TO TRUE
            MOVE 1 TO lnk-usr-fcode
            MOVE 4 TO lnk-usr-retcode
            MOVE 6 TO lnk-data-length

         WHEN customer-del-flg-true

*  User has entered a customer code and selected the DELETE
*  option to delete the customer record from the file.

            MOVE customer-c-code TO user-data-block
            SET use-rdt TO TRUE
            MOVE 2 TO lnk-usr-fcode
            MOVE 4 TO lnk-usr-retcode
            MOVE 6 TO lnk-data-length
            INITIALIZE customer-data-block

         WHEN customer-clr-flg-true

*  User has selected the CLEAR option to clear the current
*  customer details from the screen.

            SET use-rdt TO TRUE
            MOVE 0 TO lnk-usr-retcode
            INITIALIZE customer-data-block
            PERFORM Set-Up-For-Refresh-Screen
        END-EVALUATE
```

The CLEAR option uses a NULL operation because clearing the record is done locally so there is no need to contact the server. Below is an extract from the server program showing the code required to process RDT. The client/server binding sets the flag "`send-via-rdt`" so you can check the values of `lnk-usr-fcode`. On the server, the code would be similar to that shown below.

```
         WHEN send-via-rdt
            EVALUATE lnk-usr-fcode
             WHEN 1

*  For the LOAD function, the server program reads the
*  customer details from the data file, and sends the data
*  back to the client using the data area
*  customer-data-block rather than using the RDT facility.
*  Unless the RDT flag is set, the client/server bindings
*  will always pass the complete data area (defined by
*  dblksize in the configuration file) between the client
*  and the server.

                MOVE user-data-block TO customer-c-code
                SET customer-load-flg-true TO TRUE
                PERFORM... .rest of program... .

             WHEN 2
                MOVE user-data-block TO customer-c-code
                SET customer-del-flg-true TO TRUE
                PERFORM... .rest of program... .
            END-EVALUATE
```

14.10.5 Server Controlled File Management Facility

One of the problems that accompanies any client/server solution, especially as the number of clients increases, is the management of client program updates and other changes.

The override facility (see the section *Overriding Configuration File Entries*) enables you to re-route client applications while server maintenance is carried out without having to change each client's configuration file entries. The override can be either local or remote, but installing and removing a local override file on each client can be time-consuming.

The **mfcsmgr** program enables you to install and delete override files on the client using the -i and -d options respectively. See the section *Managing the Server*. The install/delete is carried out as the client program exits.

In fact, the -i option of the **mfcsmgr** program enables you to install any type of file on the client system (in the client's local directory). This means that you can use it to install new screensets or program files, enabling updates to be distributed from a central point. For security reasons, the delete option has not been similarly enhanced: it will delete an override file only; that is, a file called **mfcsovrd.cfg**.

Files to be installed using the -i option must be located on the server. If the file is in the directory from which **mfserver** was started, you need only specify the filename. If it is anywhere else, you must supply a full pathname in such a way that the filename can be extracted from the path. For example, **/u/live/update/newprog.int**, **d:\testprog.int** and **$LIVE/newtest.int** are all valid, but **$newfile** is not.

No programming changes are required to use these functions. A message is displayed informing the client that a file is being transferred and the details are recorded in the system log file.

14.11 Running the Supplied Customer Example

See the file *csbind.txt* in the **DialogSystem\demo\csbind** directory for instructions on how to run the Dialog System client/server binding demonstration.

14.12 The System Error/Message Log

Both the client and server modules of the binding maintain a log of errors and messages. All entries are date- and time-stamped and are kept in a file called **mfclisrv.log**, which is written to the directory in which the programs were started or to the directory named in the MFLOGDIR environment variable. Check this log from time to time to ensure that your system is performing correctly.

14.13 Client/Server Binding Limitations

The client/server binding imposes very few limitations, but you should be aware of the following issues:

- You can run the client/server binding modules in **.int** or **.gnt** format on Windows 95, Windows NT and UNIX platforms. In addition, on

UNIX platforms, the client/server binding modules can be linked with a user-written application program to create an executable object.

- The number of clients that can be supported is not limited by the client/server binding but may be limited by the capabilities of the server, by the network protocol, or by the performance requirements of your end users. On UNIX, for example, the limit is set by the number of sub-processes that can be spawned by **mfserver**. When a user logs onto a UNIX machine, he is assigned a unique process-id which can support a limited number of sub-processes. When clients connect via the client/server binding, they all become sub-processes of the base server. Of course, you can get around this limitation by altering the number of sub-processes permitted, or by running multiple base servers. If your UNIX machine is configured to support many hundreds of direct login users, the client/server binding running on the same machine should support the same number of clients once the sub-process limit issue has been resolved.

- The client/server binding has no recovery facilities: if the network goes down, data will be lost. Both ends of the connection will be aware of the failure and the information will be logged but that is all. The same is true of any RTS errors that cause termination of the user programs at either end of a connection. The client/server binding does not provide anything more than the standard RTS in this regard.

15 Advanced Topics

This chapter introduces some of the more advanced system features available with Dialog System.

Topics include:

- Running applications with multiple resolutions.
- How to interface to Dialog System and alternative error message files.
- How to build an interface to a file selection facility.
- How to modify menu items at run time.
- More advanced ways to use the Call Interface.
- Adding Help.

15.1 Implementing Applications to Run on Multiple Resolutions

When writing applications for production use on more than one desktop resolution size, it is possible to implement COBOL program and screenset changes which fully enable window, control and font mapping support based on the current resolution. This section covers the three areas that need to be implemented:

- Enabling the Screenset for multiple resolutions.
- Enabling font mapping.
- Setting the correct DSFNTENV environment variable using COBOL.

15.1.1 Enabling the Screenset for Multiple Resolutions

This feature is enabled by a CALLOUT to "dsrtcfg" with a flag of 9 and an identifier that tells Dialog System what resolution the screenset was DEFINED under.

The Dialog System run-time uses Panels V2 generic coordinates, which provides a basis for cross-resolution support and compatibility with differing graphics adapters. This results in differing coordinate values for what otherwise appears to be the same resolution setting.

To check the identifier you need to supply for your definition platform, a verification program is supplied with Dialog System: this displays a message box detailing the resolution identifier you need to use to implement the changes. This is executed as follows:

```
runw dsreschk
```

This program displays:

- The Panels V2 generic coordinates for the current resolution.
- A resolution identifier to be passed to the Dialog System run-time call.

Code the following dialog in your SCREENSET-INITIALIZED or other dialog table which will be executed before the windows creation:

```
CLEAR-CALLOUT-PARAMETERS $NULL

    CALLOUT-PARAMETER 1 CONFIG-FLAG $NULL
    CALLOUT-PARAMETER 2 CONFIG-VALUE $NULL
    MOVE 9 CONFIG-FLAG
    MOVE resolution id CONFIG-VALUE
    CALLOUT "dsrtcfg" 3 $PARMLIST
```

`CONFIG-FLAG` and `CONFIG-VALUE` should be C5 4 byte data fields.

Once implemented, all windows, dialog boxes and their child controls will be resized in proportion to the current resolution. See the topic *Multiple Resolution and Dynamic Window Sizing* in the Help.

15.1.2 Enabling Font Mapping

In addition to enabling multiple resolution support in each of your screensets, the objects you create need to be allocated a font style so that their font used at run time may be adjusted depending upon the resolution on which your production application runs.

When you specify a font using **Fonts** on the **Edit** menu, you can specify both a typeface and a style name for that font. The style name is a user-defined name that represents the specified typeface, pointsize and attributes. This style name enables your fonts to be increased or decreased in size appropriate to resolutions used at run time.

Select the font typeface, pointsize and attributes required, and enter your required style name in the Selection box provided. For example, My-Style.

Click **Apply** to apply the selected font style to the current object (or group of objects).

When you specify a binary font side file (with a **.dfb** extension) using **Resource Files** on the **Options** menu, Dialog System looks up the style name in the binary font side file at run time and uses the details of that font for the resolution platform in use. If the style is not found in the side file, the default font is used.

Assume that you want to use a style created under 1024*768 resolution and transfer that to the 640*480 resolution. You have specified a binary font side file (with a **.dfb** extension) and saved the screenset. When you save the screenset, a textual version of the font side file (**.dft** file) is created (or appended to) and contains:

```
[NT]

FONT-RECORD
    STYLENAME My-Style
    ATTRIBUTES BITMAPPED, PROPORTIONAL
    POINTSIZE 12
    TYPEFACE "Roman"
END-RECORD
```

In order to transfer the screenset and binary font side file to the production environment, you need to edit the **.dft** side file to incorporate a new definition for running under each possible resolution. The file now contains:

```
[RESOLUTION1]
FONT-RECORD
    STYLENAME My-Style
    ATTRIBUTES BITMAPPED, PROPORTIONAL
    POINTSIZE 12
    TYPEFACE "Roman"
END-RECORD

[RESOLUTION2]
FONT-RECORD
    STYLENAME My-Style
    ATTRIBUTES BITMAPPED, PROPORTIONAL
    POINTSIZE 8
    TYPEFACE "Roman"
END-RECORD
```

Notes:

- The section marker NT has been altered to be appropriate to the definition resolution.
- RESOLUTION2 was created by editing the file and reducing the pointsize used.

Having added the new section marker and attributes, you need to convert the side file to binary format:

```
run dsfntgen /t appstyle.dft /b appstyle.dfb /c
```

This creates the binary side file. Now you need to set the environment variable:

```
set DSFNTENV=RESOLUTION2
```

Dialog System automatically selects the new font attributes for that font style definition. DSFNTENV is set to RESOLUTION2 under 640*480 resolutions where smaller fonts are required as there is less available display space.

If DSFNTENV is not set, then no mapping is performed and the default values, stored in the screenset, are used. Provided your screensets are consistent in use of style names you can use just one font side file for applications consisting of many screensets.

15.1.3 Setting the DSFNTENV Environment Variable Using COBOL

You can place the setting of the required DSFNTENV environment variable under control of your COBOL program. This section explains how this is achieved, but you first need to determine the resolution that the application is currently running in.

Do this by using the following code:

```
* Determine resolution & set DSFNTENV accordingly
     MOVE LOW-VALUES            TO P2I-Initialization-Record
     MOVE P2I-Current-Environment  TO P2I-Environment
     MOVE 0                        TO P2I-Name-Length
     MOVE Pf-Initialize            TO P2-Function
     CALL "PANELS2" USING P2-Parameter-block
                          P2I-Initialization-Record
     END-CALL
     IF  P2-Status NOT = 0
         DISPLAY "Warning: Unable to start PANELS2. " &
                 "Error No = "
                 P2-STATUS
         STOP RUN
     END-IF.
```

You should include the copyfiles **pan2link.cpy**, and **pan2err.cpy** in your program's Working-Storage section to obtain the required variables and Panels V2 interface record.

You can then test the values returned in `P2I-Screen-Width` and `P2I-Screen-Height`, and using the following code and COBOL reserved words, set the environment variable to point to the font side file section marker required.

```
     IF P2I-Screen-width = 640
     AND P2I-Screen-Height = 480
         DISPLAY "DSFNTENV" UPON ENVIRONMENT-NAME
         DISPLAY "RESOLUTION2" UPON ENVIRONMENT-VALUE
     END-IF
     IF P2I-Screen-width = 1024
     AND P2I-Screen-Height = 768
         DISPLAY "DSFNTENV" UPON ENVIRONMENT-NAME
         DISPLAY "RESOLUTION1" UPON ENVIRONMENT-VALUE
     END-IF
```

This environment variable should be established before your first call to Dialog System and will remain in existence until termination of the COBOL run unit.

It is possible to refine targeted production platform resolutions further (for example, 800*600 - Large fonts) by adding the following code BEFORE the call to Panels V2:

```
ADD P2I-Generic-Coordinates TO P2I-Environment
```

This will alter the returned coordinates to be in line with the values returned by the DSRESCHK program. You then need to adjust your subsequent IF statements to reflect the supported production resolutions.

You have now implemented multiple resolution support in your screenset, enabled all screenset object fonts to be remapped at run time, and established the correct environment variable according to the current production resolution platform.

Provided the layout of your windows has been well designed, your screenset should now be fully portable across different resolutions.

15.2 Using the Dialog System Error Message File Handler

Your calling program can display error messages using the Dialog System error message file handler and its display capabilities. You might want to do this if you need to perform some complex validation that Dialog System is not designed to perform.

The error message is stored in an error message file as normal. The simplest method is to use the error message file that the screenset is using. Alternatively, you can use one or more other error message files, but you must explicitly open and close them.

To use the Dialog System error message file handler at run time, your program needs to:

- Ensure that the error file has been opened.

- Extract the error message.
- Pass the error message to Dialog System with instructions to display it.

This fragment of code illustrates the procedures:

```
 1 working-storage section.
 2 ...
 3 01  error-file-linkage.
 4     03 short-file-name       pic x(8).
 5     03 file-access           pic xx.
 6     03 filler                pic x(6).
 7     03 errhan-code           pic xx.
 8
 9 01 error-file-data.
10     03 error-record-number   pic 9(4) comp.
11     03 error-record-contents pic x(76).
12 ...
13 procedure division.
14     ...
15     initialize ds-control-block
16     initialize data-block
17     move "N" to ds-control
18     move data-block-version-no to ds-data-block-version-no
19     move version-no to ds-version-no
20     move "custom" to ds-set-name
21     call "dsgrun" using ds-control-block
22                         data-block
23     ...
24     move "E" to ds-control
25     call "dsgrun" using ds-control-block
26                         data-block
27     ...
28     move "CUSTERR" to short-file-name
29     move "R" to file-access
30     move 101 to error-record-number
31     call "dserrhan" using error-file-linkage
32                           error-file-data
33     ...
```

Lines 3-11:

```
01  error-file-linkage.
    03 short-file-name       pic x(8).
    03 file-access           pic xx.
    03 filler                pic x(6).
    03 errhan-code           pic xx.

01 error-file-data.
```

```
     03 error-record-number   pic 9(4) comp.
     03 error-record-contents pic x(76).
```

You need to define these records in the Working-Storage Section.

Lines 15-22:

```
     initialize ds-control-block
     initialize data-block
     move "N" to ds-control
     move data-block-version-no to ds-data-block-version-no
     move version-no to ds-version-no
     move "custom" to ds-set-name
     call "dsgrun" using ds-control-block
                         data-block
```

Initial call to Dialog System.

Lines 24-26:

```
     move "E" to ds-control
     call "dsgrun" using ds-control-block
                         data-block
```

If you intend to use the error message file the screenset uses, you must make sure that it is open by calling Dialog System in the normal way, but with the parameter "E" in ds-control. Dialog System immediately returns control to the program having opened the file and replaces the parameter "C" (ds-continue) in ds-control.

ds-err-file-open is a pre-defined value for ds-control. It is defined in the Control Block as:

```
         05 ds-err-file-open            pic x value "E".
```

Thus, an alternative to line 24 is:

```
     move ds-err-file-open to ds-control
```

Line 28:

```
     move "custerr" to short-file-name
```

Moves the name of the error file (not including the extension .err) to the data item short-file-name.

Line 29:

```
     move "R" to file-access
```

"R" is for Read.

Line 30:

```
      move 101 to error-record-number
```

Moves the number of the error message you want to display.

Lines 31-32:

```
      call"dserrhan" using error-file-linkage
                           error-file-data
```

This call reads the error message file. If the call is successful, it returns with errhan-code equal to "OK" and the error message in error-record-contents. If the file is not found, the value "NF" (Not Found) is placed in errhan-code. If the file is in use or if you tried to open more than 16 files, the value "FU" (File in Use) is placed in errhan-code. You do not need to close the error file because Dialog System does this automatically.

You now need to put the error message in a data item in the Data Block so Dialog System can display it. One way to do this is to set up a procedure that displays a message box, using the data item as its argument, then call Dialog System requesting that procedure.

For example, if you have a message box named ERR-DISPLAY-MB, define a procedure with the following dialog:

```
DISPLAY-ERR-MSG
  INVOKE-MESSAGE-BOX ERR-DISPLAY-MB ERR-MSG-EF $REGISTER
```

Then in your program, request this procedure when you return to Dialog System with a statement such as:

```
      move "display-err-msg" to ds-procedure
```

where display-err-msg is the name of the procedure and ds-procedure is a data item in the Control Block. On entry to Dialog System, this procedure is executed.

15.2.1 Using an Alternative Error Message File

To use an error message file other than the one specified by your screenset, at run time your program needs to:

- Open the error file.
- Extract the error message.

- Pass the error message to the Dialog System run-time system with instructions to display it.
- Ensure that any error files not in use by the screenset are closed.

The only difference between using an alternative message file and the Dialog System error message file is that you must explicitly open and close the alternative file.

The following fragment of code illustrates the procedures:

```
 1 working-storage section.
 2 ...
 3 01  error-file-linkage.
 4     03 short-file-name       pic x(8).
 5     03 file-access           pic xx.
 6     03 filler                pic x(6).
 7     03 errhan-code           pic xx.
 8
 9 01 error-file-data.
10     03 error-record-number   pic 9(4) comp.
11     03 error-record-contents pic x(76).
12 ...
13 procedure division.
14     ...
15     move "ALTERR" to short-file-name
16     move "NI" to file-access
17     move "C:\CUST\ALTERR.ERR" to error-file-data
18     call "dserrhan" using error-file-linkage
19                           error-file-data
20     ...
21     move "ALTERR" to short-file-name
22     move "R" to file-access
23     move 101 to error-record-number
24     call "dserrhan" using error-file-linkage
25                           error-file-data
26     ...
27     move "ALTERR" to short-file-name
28     move "NC to file-access
29     call "dserrhan" using error-file-linkage
30                           error-file-data
31     ...
```

Lines 3-11:

```
01  error-file-linkage.
     03 short-file-name       pic x(8).
     03 file-access           pic xx.
     03 filler                pic x(6).
     03 errhan-code           pic xx.
```

```
 01 error-file-data.
     03 error-record-number   pic 9(4) comp.
     03 error-record-contents pic x(76).
```

Again, you need to define these records in the Working-Storage Section.

Line 15:

```
     move "ALTERR" to short-file-name
```

Identifies the alternative file.

Line 16:

```
     move "NI" to file-access
```

"NI" is the file access code to open a new file .

Line 17:

```
     move "C:\CUST\ALTERR.ERR" to error-file-data
```

Specifies the path and name of the error file to be opened.

Lines 18-19:

```
     call "dserrhan" using error-file-linkage
                           error-file-data
```

Reads the error file with the call statement. If the call is successful, it returns with `errhan-code` equal to "OK" and the error message in error-record-contents. If the call is unsuccessful, `errhan-code` is "NF" for Not Found. If the file is already open or you try to open more than 16 files, `errhan-code` is "FU" for "File in Use".

Line 27:

```
     move "ALTERR" to short-file-name
```

Specifies the file to close.

Line 28:

```
     move "NC to file-access
```

"NC" is the close file code.

Lines 29-30:

```
     call "dserrhan" using error-file-linkage
                           error-file-data
```

This call closes the file. `errhan` closes the file and returns to the program.

You now need to display the error message. Refer to the section *Using the Dialog System Error Message File Handler*above for a method to do this.

15.3 Building an Interface to a File Selection Facility

When the user needs to enter the name of a file, it is useful to provide a prompt that displays the files currently available on the system. The user can then browse around the drives and directories on his system and select an existing filename or enter the name of a new one.

For example, Dialog System uses this file selection facility when you generate a copybook for a screenset and specify the filename to use.

This section describes how to provide this facility from your Dialog System applications using the Dsdir Dialog System extension. Dialog System Extensions (DSX) is the term given to dialog functions implemented by using the CALLOUT dialog function. Dialog System extensions are supplied to enable you to perform many regular programming tasks, such as calling on-line help or providing a file selection facility. For more information about Dialog System extensions, see the topic *Dialog System Extensions* in the Help.

15.3.1 The Dirdemo Sample Screenset

A sample application, Dirdemo, illustrates how you can use the Dsdir Dialog System extension in your applications. You can run the application to see the facilities that are available with the Dsdir Dialog System extension.

You can run the Dirdemo sample directly from the Dialog System definition software. There is no COBOL program with this sample. Load the screenset **dirdemo.gs** and run it through the Screenset Animator. For information on running a screenset, see the chapter *Using the Screenset*. Alternatively, you can run the Dirdemo sample using Dsrunner. See the chapter *Multiple Screensets* for more information on Dsrunner.

The program displays two prompts:

- A filename.

  If you enter text into the filename field, initially only files matching that filename are displayed. You can include the "*" and "?"

wildcard characters in the filename. For example, enter ***.gs** to see a list of all Dialog System screensets. If you leave this field blank, a default of *.* is provided.

- A window title.

  The contents of the window title field are displayed as the title of the file selection facility. For more information, see the description of Dsdir in the topic *Dialog System Extensions* in the Help.

To use the file selection facility window select one of the options on the pulldown menu:

Open — Select a file to be opened. The user can select only a file that already exists.

Save — Select a file to be saved. The user can select any file that already exists (or supply the name of a new file).

Check — Check whether or not the file entered in the filename field actually exists. When this function is selected, wildcard characters are not accepted in the filename field.

Exit — Exit.

None of the options in this menu actually opens a file. Once the user has selected a file, you can decide which function to use.

15.3.2 The Dirdemo Data Block

To call the Dsdir Dialog System extension, you must define the following in the screenset's Data Block, in addition to any other data that the screenset uses:

```
DSDIR-PARAMS                      1
    DSDIR-FUNCTION               X    4.0
    DSDIR-RETURN-CODE            C    2.0
    DSDIR-FILENAME               X  256.0
```

To set your own title for the file selector window, you also need to add the following to the screenset's Data Block:

```
DSDIR-PARAMS2
    DSDIR-TITLE                  X  256.0
```

These fields are described in the section about the Dsdir Dialog System extension in the topic *Dialog System Extensions* in the Help.

15.3.3 The Dirdemo Dialog

The Dirdemo sample calls the Dsdir Dialog System extension whenever a file is required to be selected.

The dialog (attached to the main window) to do this is:

```
1      DO-DSDIR-CALL
2        CLEAR-CALLOUT-PARAMETERS $NULL
3        CALLOUT-PARAMETER 1 DSDIR-PARAMS $NULL
4        CALLOUT-PARAMETER 2 DSDIR-PARAMS2 $NULL
5        CALLOUT "Dsdir" 0 $PARMLIST
6 ...
17     @OPEN-PD
18       MOVE "open" DSDIR-FUNCTION(1)
19       EXECUTE-PROCEDURE DO-DSDIR-CALL
20     @SAVE-PD
21       MOVE "save" DSDIR-FUNCTION(1)
22       EXECUTE-PROCEDURE DO-DSDIR-CALL
23     @CHECK-PD
24       MOVE "chek" DSDIR-FUNCTION(1)
25       EXECUTE-PROCEDURE DO-DSDIR-CALL
```

Line 1:

```
    DO-DSDIR-CALL
```

This procedure makes the call to the Dsdir Dialog System extension.

Lines 2-4:

```
      CLEAR-CALLOUT-PARAMETERS $NULL
      CALLOUT-PARAMETER 1 DSDIR-PARAMS $NULL
      CALLOUT-PARAMETER 2 DSDIR-PARAMS2 $NULL
```

Defines the parameters required for the CALLOUT. Because more than one parameter is required when the file selector window title is specified, all the parameters must be defined before you make the CALLOUT. The `CLEAR-CALLOUT-PARAMETERS` function is used to ensure that any previously defined parameters are removed before the new parameters are defined.

Line 5:

```
     CALLOUT "Dsdir" 0 $PARMLIST
```

This line calls the Dialog System extension using the previously defined parameters. The file selector window is displayed, and the user can select a file. When the user has selected a file, or cancelled the file selection, control is returned to your screenset.

Line 17:

```
@OPEN-PD
```

The user selected the **Open** option on the pulldown menu.

Line 18:

```
  MOVE "open" DSDIR-FUNCTION(1)
```

This line sets up the parameter that tells the Dialog System extension to allow the user to select a file to open. In this mode, the user can only select files that already exist.

Line 19:

```
  EXECUTE-PROCEDURE DO-DSDIR-CALL
```

Carries out the procedure that calls the Dialog System extension.

Lines 20-22:

```
@SAVE-PD
  MOVE "save" DSDIR-FUNCTION(1)
  EXECUTE-PROCEDURE DO-DSDIR-CALL
```

The user selected the **Save** option on the pulldown menu. In save mode, the user can select any file, or specify the name of a new file.

Lines 23-25:

```
@CHECK-PD
  MOVE "chek" DSDIR-FUNCTION(1)
  EXECUTE-PROCEDURE DO-DSDIR-CALL
```

The user selected the **Check** option on the pulldown menu. The chek function requests the Dialog System extension to check for the existence of the supplied filename. The file selector window is not displayed.

15.4 Modifying Menu Items at Run Time

As well as defining menu items at definition time, you can define dialog to add, delete, or change menu items at run time. For example, you can add a list of recently opened files to your menu, or a list of open windows in a Multiple Document Interface (MDI) application.

All the options for defining a menu bar at definition time can also be used at run time, except that you can only add new items to an existing menu bar, and you cannot add menu choices to context menus.

You add menu choices to a menu bar using the ADD-MENU-CHOICE dialog function. One of the parameters for this function is a text string that specifies all the options for the menu item. The following example shows all the options specified in their correct places, although in practice, you should never define a menu item like this:

```
GET-MENU-CHOICE-REFERENCE MAINWIN EDITMENU PARENT-REF
ADD-MENU-CHOICE PARENT-REF "~item>" $EVENT-DATA
GET-MENU-CHOICE-REFERENCE MAINWIN $EVENT-DATA PARENT-REF
ADD-MENU-CHOICE PARENT-REF "*~choice@item&ctrl+c" $EVENT-DATA
```

Where:

PARENT-REF This parameter specifies which menu (or sub-menu) this new item will be added to. The value of this parameter must be obtained using the GET-MENU-CHOICE-REFERENCE function. The value can never be zero.

* The asterisk causes the menu item to be created with the checkmark type set to off (the default is none). The asterisk must appear as the first character of the text parameter, otherwise it will be assumed to be part of the menu text. If you intend the asterisk to be part of the menu text, add a space before it.

~ The tilde causes the next character to be defined as the mnemonic character for the menu choice. You are not warned if you define menu choices with duplicate mnemonic characters (unlike when defining a menu choice at definition time). The tilde character can appear anywhere in the choice text.

> The greater than symbol, if it appears as the last character of the text parameter, signifies that when the menu item is selected, a submenu of choices is to be displayed. If you specify that a menu choice is to display a submenu, the checkmark state and shortcut key settings are ignored.

choice The text up to the @ (at) defines the text that is displayed in the menu bar (the choice text).

@item — The '@' signifies that the next text (up to the next @, space, & or >) is the name of the menu item. Normal Dialog System object naming rules apply to the menu item name. If there is a procedure defined in the screenset with the same name as the menu item, selecting the menu item causes the procedure with that name to be executed. The menu item name is optional.

&ctrl+c — The ampersand signifies that the next text (up to the next @, space, & or >) is the shortcut key for the menu item. You can specify the shortcut key either in the format you use when you define a menu choice in menu bar definition (such as CTC), or in the format that the shortcut key will be displayed on screen (for example, Ctrl+C).

When you create a menu bar item using this function, the ID of the created menu bar item is returned. You can then use this ID with functions such as DISABLE-MENU-CHOICE to manipulate the menu item further.

You can delete menu items using the DELETE-MENU-CHOICE function. The items that you delete can be menu items defined in menu bar definition, or menu items added using the ADD-MENU-CHOICE function.

Also, you can update the text of menu items by using the UPDATE-MENU-TEXT function.

See the relevant functions in the Help for more details, including examples of using the functions.

15.5 Using the Call Interface

The Dialog System call interface can be used by your calling program to provide more sophisticated ways of using Dialog System. For example:

- You can use multiple screensets.
- You can use multiple instances of the same screenset.

Using these features, you can divide your user interface into logical components that are used as required, use multiple copies of the same screenset, and group all your error messages into a single file. See the chapter *Using the Screenset* for details.

15.6 Adding Help

This section discusses the Helpdemo screenset in detail. The Helpdemo screenset:

- Is located in **DialogSystem\demo\helpdemo**.
- Uses the Dsonline Dialog System extension - the Dialog System extension that displays Windows help.
- Uses Windows Help to display context-sensitive help.
- Provides access to all the facilities of the Help system.
- Is supplied as a sample screenset.

You might want to run the screenset while you read this section.

15.6.1 Running the Helpdemo Sample

You can run the Helpdemo sample directly from the Dialog System definition software. There is no COBOL program with this sample.

Load the screenset **helpdemo.gs** and run it through the Screenset Animator.

For information on running a screenset, see the section *Testing the Screenset* in the chapter *Using the Screenset*.

Alternatively, you can run the Helpdemo sample using Dsrunner. See the chapter *Multiple Screensets* for more information on Dsrunner.

15.6.2 The Helpdemo Data Block

To call the Dsonline Dialog System extension, you must define the following in the screenset's Data Block, in addition to any other data that the screenset uses:

```
DSONLINE-PARAMETER-BLOCK             1
    DSONLINE-FUNCTION              X    18.0
    DSONLINE-RETURN                C    2.0
    DSONLINE-HELP-FLAGS            C    2.0
    DSONLINE-HELP-CONTEXT          9    18.0
    DSONLINE-HELP-TOPIC            X   32.0
    DSONLINE-HELP-FILE             X  256.0
```

The meaning of each of these fields is given in the description of the Dsonline Dialog System extension in the topic *Dialog System Extensions* in the Help.

15.6.3 The Helpdemo Dialog

The Helpdemo sample calls the Dsonline Dialog System extension whenever help information is required. Help might be required for a particular field, or when the user selects an option on the **Help** menu.

The dialog (attached to the main window) to do this is:

```
F1
    MOVE 1 DSONLINE-HELP-CONTEXT(1)
    BRANCH-TO-PROCEDURE CONTEXT-HELP
  @HELP-CONTENTS
    MOVE "cont" DSONLINE-FUNCTION(1)
    BRANCH-TO-PROCEDURE CALL-ON-LINE
  @HELP-INDEX
    MOVE "indx" DSONLINE-FUNCTION(1)
    BRANCH-TO-PROCEDURE CALL-ON-LINE
  @EXIT-F3
    SET-EXIT-FLAG
    RETC
  CONTEXT-HELP
    MOVE "ctxt" DSONLINE-FUNCTION(1)
    BRANCH-TO-PROCEDURE CALL-ON-LINE
  CALL-ON-LINE
    MOVE "helpdemo.hlp" DSONLINE-HELP-FILE(1)
    MOVE 0 DSONLINE-HELP-FLAGS(1)
    CALLOUT "dsonline" 0 DSONLINE-PARAMETER-BLOCK
```

Line 1:

```
F1
```

The user pressed the **F1** key. Every entry field in this example has individual control dialog for the event caused by the **F1** key, so if this dialog processes the event, it means that the focus is not on one of the entry fields. When this event is processed, general help is provided.

Line 2:

```
    MOVE 1 DSONLINE-HELP-CONTEXT(1)
```

Set the context number to be displayed. The context numbers are defined in the On-line help file. If you do not specify context numbers, the On-line help file builder (ohbld) provides them for you. In this example, `1` is the context number for general help.

Line 3:

```
    BRANCH-TO-PROCEDURE CONTEXT-HELP
```

Branches to the procedure that displays the help. Because all context help is displayed in the same way, a procedure is used, to avoid repeating dialog.

Line 4:

```
  @HELP-CONTENTS
```

The user selected **Contents** on the **Help** menu.

Line 5:

```
    MOVE "cont" DSONLINE-FUNCTION(1)
```

The Dsonline Dialog System extension can do several different things. To tell it what to do, you must provide it with a function name (a four character string). The `cont` function requests that the contents of the On-line help file are displayed.

Line 6:

```
    BRANCH-TO-PROCEDURE CALL-ON-LINE
```

Branches to the procedure that calls the Dialog System extension.

Lines 7-9:

```
  @HELP-INDEX
    MOVE "indx" DSONLINE-FUNCTION(1)
    BRANCH-TO-PROCEDURE CALL-ON-LINE
```

The user selected **Index** on the **Help** menu. The `indx` function requests that the index for the On-line help file be displayed.

Line 10:

```
@EXIT-F3
```

The user selected **Exit** on the **File** menu, or pressed the **F3** key.

Line 11:

```
    SET-EXIT-FLAG
```

The exit flag is a flag that Dsgrun returns to the calling program when a RETC is executed. The exit flag tells Dsrunner that the screenset has finished processing.

Line 12:

```
    RETC
```

Return to the calling program (in this example, either Dialog System or Dsrunner).

Lines 13-15:

```
CONTEXT-HELP
    MOVE "ctxt" DSONLINE-FUNCTION(1)
    BRANCH-TO-PROCEDURE CALL-ON-LINE
```

This procedure is used to display context help. The `ctxt` function requests that context-sensitive help be displayed. The `CALL-ON-LINE` procedure is used to call the Dialog System extension.

Lines 16-19:

```
CALL-ON-LINE
    MOVE "helpdemo.hlp" DSONLINE-HELP-FILE(1)
    MOVE 0 DSONLINE-HELP-FLAGS(1)
    CALLOUT "dsonline" 0 DSONLINE-PARAMETER-BLOCK
```

This procedure calls the Dsonline Dialog System extension. It sets up the name of the help file in the parameter block, and turns off all the flag settings for the On-line Help system (see the Dsonline Dialog System extension in the topic *Dialog System Extensions* in the Help for information about the flags).

15.6.4 Entry Field Dialog

As well as the dialog on the main window, each entry field in the window has dialog. Each entry field sets a different context number, but otherwise, the dialog is the same.

For the Product Code entry field, the dialog is:

```
F1
    MOVE 2 DSONLINE-HELP-CONTEXT(1)
    BRANCH-TO-PROCEDURE CONTEXT-HELP
```

This dialog is similar to that for the **F1** key on the main window, but uses a different context number, where `2` is the context number for help about this entry field.

15.7 Further Information

See the chapters *Using the Screenset* and *Multiple Screensets* for further information about controlling the use of screensets and using multiple screensets and multiple instances of screensets.

The chapter *Multiple Screensets* also contains a detailed description of the call interface. The topic *The Call Interface* in the Help also provides information on the Control Block, including the Event Block, the Data Block, use of the Screenset Animator, version checking, and the values the calling program returns to Dialog System.

16 Questions and Answers

This chapter lists and answers questions that are frequently raised with the Technical Support.

Note: Some of the questions relate to all environments. Some are specific to particular environments. The notation in the left margin indiactes which environment is affected.

The Checker rejects my Data Block that I generated directly from my screenset. What is going wrong?

If you have written your program so that it expects to have all data items in the Data Block prefixed with the screenset ID, you must generate the screenset that way.

For example, the Customer program has the CUSTOMER prefix on all the data items in the Data Block. If you generate the Data Block and do not specify you want the Data Block prefixed with the screenset ID, the data items are not prefixed by CUSTOMER and the checker rejects any reference to the items.

How can I use the Screenset Animator facilities from my program without recompiling?

You have two options:

- Make the name of the Dialog System run-time system a variable. For example, the Customer demo can be changed to use the following level-01 data item:

  ```
  01 dialog-system    pic x(8) value "dsgrun".
  ```

  Then, while debugging your application, you can change the value of `dialog-system` to "ds". ds is the Dialog System run-time system that enables you to use the Screenset Animator.

Once you have done this, all future calls to the Dialog System run-time will use Screenset Animator (even if you change the value of `dialog-system` back to dsgrun).

- Use the "T" and "O" options in `ds-control`. Setting `ds-control` to "T" turns the Screenset Animator on. Setting `ds-control` to "O" turns the Screenset Animator off.

  To turn Screenset Animator on:

  a Set a breakpoint on the Dialog System CALL statement.

  b Change the value of the data item `ds-control` to "T".

  c Step on the call to Dsgrun.

  The "T" option causes Dialog System to issue an internal call to turn the Screenset Animator facility on and exit.

  d Reset your cursor on the Dialog System CALL statement.

  e Zoom your program again. Screenset Animator will appear the next time a line of dialog is executed.

  To turn Screenset Animator off:

  a Change the value of the data item `ds-control` to "O".

  b Step on the call to Dsgrun.

  Screenset Animator will disappear. As with the "T" option, the "O" option causes Dialog System to issue an internal call to turn the Screenset Animator facility off and then exit.

  c Reset your cursor on the Dialog System CALL statement.

  d Zoom your program. Screenset Animator is turned off.

Why is a dialog box not displayed when I use a SHOW-WINDOW function?

This happens if the dialog box is a modal dialog box, because SHOW-WINDOW does not paint this type of dialog box on the screen. A modal dialog box is shown only when the focus is set on it using the SET-FOCUS function. SHOW-WINDOW does paint a window or a modeless dialog box on the screen.

How do I validate fields as soon as they lose focus?

The most common problem with implementations of field-by-field validation is that the application usually ends up in an infinite loop of validation errors. For example, if an entry field loses focus and is validated and the validation fails, focus is returned to the object. However, this causes another control to lose focus, and if that control is also validated, an infinite loop can result.

Another problem is that you might sometimes not want to validate a field. For example, if a user clicks **Cancel** or **Help** in a dialog box, or switches to a different application.

Here is one way to implement field-by-field validation which avoids the infinite looping problems, and allows you to implement **Cancel** and **Help** buttons:

1 Define a flag in your data block:

```
VALIDATION-ERROR-FOUND      C5    1.0
VALIDATION-ERROR-FIELD      C5    2.0
VALIDATION-ERROR-MESSAGE    C5   80.0
```

If you already have an error message field defined in your screenset, you can use that field instead of defining VALIDATION-ERROR-MESSAGE. Otherwise define the VALIDATION-ERROR-MESSAGE field as the error field.

2 Add the following global dialog:

```
REPORT-VALIDATION-ERROR
* Report a validation error
     INVOKE-MESSAGE-BOX VALIDATION-ERROR-MBOX
                        VALIDATION-ERROR-MESSAGE $REGISTER
     SET-FOCUS VALIDATION-ERROR-FIELD
     TIMEOUT 1 CLEAR-VALIDATION-ERROR-TIMEOUT
   CLEAR-VALIDATION-ERROR-TIMEOUT
* Reset the validation error flag and TIMEOUT
     MOVE 0 VALIDATION-ERROR-FOUND
     TIMEOUT 0 $NULL
   DO-NOTHING
* Do-nothing procedure. Contains no functions.
```

The REPORT-VALIDATION-ERROR procedure is executed after a short timeout. It reports a validation error that has been detected, then sets focus on the field that is in error. Another timeout is then set to execute the CLEAR-VALIDATION-ERROR-TIMEOUT once all the focus events have been processed. This event clears the flags associated with the validation failure.

3 Add the following to SCREENSET-INITIALIZED dialog:

```
MOVE 0 VALIDATION-ERROR-FOUND
```

This clears the VALIDATION-ERROR-FOUND flag ready for the first validation error.

4 Add the following line of dialog as the first line of LOST-FOCUS dialog for any objects which have LOST-FOCUS dialog and which could cause a focus change:

```
IF= VALIDATION-ERROR-FOUND 1 DO-NOTHING
```

5 In all places where you have dialog which causes a focus change due to a validation error, add the following dialog immediately before the dialog that causes the focus change:

```
MOVE $EVENT-DATA VALIDATION-ERROR-FIELD
     MOVE 1 VALIDATION-ERROR-FOUND
     TIMEOUT 1 REPORT-VALIDATION-ERROR
```

This dialog example assumes that $EVENT-DATA is set to the object ID of the object that failed validation. $EVENT-DATA is set to the object ID when a VAL-ERROR occurs.

6 If you want to cancel a validation error before it is reported, add the following dialog where you want the error to be canceled:

```
EXECUTE-PROCEDURE CLEAR-VALIDATION-ERROR-TIMEOUT
```

If you want to cancel a validation error on a push button, you need to cancel the error on a GAINED-FOCUS event on the button, not the BUTTON-SELECTED event. A timeout can happen between the GAINED-FOCUS and BUTTON-SELECTED events, which would trigger the validation error message.

It is also particularly important that you cancel any validation error when your application loses focus. Otherwise you will prevent users from switching to a different application and you will also interfere with the operation of Screenset Animator and the Net Express IDE. To cancel any validation error when your application loses focus, implement LOST-FOCUS dialog on the window. Note that this also applies when switching focus to other windows in your application.

When a validation error causes the focus to be changed, this sets both the VALIDATION-ERROR-FOUND flag and a very short TIMEOUT to clear the flag again. Since TIMEOUT events do not happen if there are any events still to be processed, the TIMEOUT event is guaranteed to happen after the LOST-FOCUS event. If the LOST-FOCUS event finds the VALIDATION-ERROR-FOUND flag set, then it doesn't validate the field that lost focus (and so doesn't cause a focus change).

How can I get more information about a Dialog System run-time error?

When an error occurs in your application, the Dialog System run-time returns an error code (DS-ERROR-CODE), and two additional error details (DS-ERROR-DETAILS-1 and DS-ERROR-DETAILS-2), which allow you to determine what went wrong with your application.

However, particularly with error code 17, you need to look up information in several different tables in order to get all the information available about the error. Also, some of the other error codes use the error details to return more information about the error, but you have to refer to the documentation in order to decode these.

Dialog System contains an error reporting module, **dsexcept.dll**, which can decode the error codes for you, and can also provide additional information that is not available from the error codes alone.

In order for **dsexcept.dll** to be active, you need to make sure that it is in a directory pointed to by the $COBDIR environment variable. The **dsexcept.dll** file can be found in your Dialog System **\bin** directory.

Notes:

- **dsexcept.dll** does not provide as much information for run-time format screensets.

  This is to prevent users of your production applications attempting to debug or reverse-engineer your screensets.

- Screenset Animator disables the **dsexcept.dll** module.

How much memory will my screenset take up at run time?

It is impossible to determine exactly how much memory a screenset will take up at run time. It depends on several factors, including the amount of free memory you have available. However, in general the amount memory used by a screenset is not particularly large.

All screensets are held in virtual files that are paged out to disk as memory demands increase. The only memory that each screenset is guaranteed to take up is 512 bytes required by the COBOL run-time system (RTS) to manage the virtual files.

Each virtual file opened will require the 512 bytes (unless it is more than 64K bytes in which case it will require an extra 512 bytes). This means that if Dialog System uses one virtual file for each screenset then the minimum would be 512 bytes for each screenset.

If there is sufficient memory, the RTS will hold as many of the virtual files in memory as it can. When there is insufficient memory, the RTS starts to page the virtual files to disk.

Hence the memory taken up by the screenset is variable, but should never be excessive.

For more information, see the topic *Dialog System Limits* in the Help.

How do I add color to a text field?

The Dialog System Text object is simply text painted on your window and for this reason is not configurable. It is possible, however, to achieve the effect you require through the use of an Entry Field object. Select the Display-only property of this field, and define its initial text (or master field value) to achieve the same results, but with the colors configurable via the normal means.

How do I make screensets containing list-boxes display more quickly?

If your list-box is tied to a master field group, it is possible to reduce load time by temporarily adjusting the internally recognized size of that group using the SET-DATA-GROUP-SIZE function. This tells Dialog System that internally it should recognize only (n) items in that group array - preventing data in occurrences beyond that figure being inserted into objects. However, the calling program can access all the data items in the group, and you may use the same function to reset the internal size when appropriate.

Why does my application appear to enter a controlled loop when using multiple screensets?

You might experience this when using DS-PUSH-SET and DS-USE-SET combinations in router-based applications. Specifically, the effect is usually attributed to coded dialog script, which forces the application into an event-based loop.

When using multiple screensets in this manner, you use the following dialog to re-activate a previous screenset:

```
OTHER-SCREENSET
 MOVE 1 OTHER-SET-EVENT-FLAG
 REPEAT-EVENT
 RETC
```

Your COBOL program then makes a decision based on the DS-EVENT-SCREENSET-ID to reload the screenset on which the event occurred.

The error described is usually caused by setting a value in DS-PROCEDURE, which causes execution of that dialog table in the reloaded screenset. The executed procedure then carries out its own functions, often causing more events in the original screenset, and results in the effect being experienced.

The reason for this effect is explained in the chapter *Multiple Screensets*, in the section *The Sequence of Events*.

DS-PROCEDURE should not be set when re-loading a screenset in response to an OTHER-SCREENSET event.

Why, when my selection box has the auto-insert property set to off, can list items become duplicated in the dropdown list?

This is primarily caused by inappropiate use of this control.

Appropriate use of a selection box control is for the selection of an item from a pre-defined list. If required, the user-selected item can be placed in an associated Master field, to be used by the application as appropriate.

The important factor here is that the dialog functions which populate a selection box insert the list items, delimited by the hexadecimal value "0A". You cannot insert items into a selection box from a Master field group.

Duplication occurs when the text in the associated Master field contains a value other than that inserted into the list (that is, is SPACE delimited) and is not delimited by X"0A". Duplication appears to occur but the items are not delimited in the same way.

Incorrect population of the associated Master field, and the REFRESH-OBJECT function are prime mechanisms causing this undesired behavior.

How do I control the tabbing order of controls?

Select the **Edit**, **Controls** menu option and re-order the displayed list as required. See the topic *Controls* in the Help.

Compiling 256-color bitmap resources into a .dll will not work.

256-color bitmaps must be specified as separate files to be included in a Dialog System application, and the MFDSSW /B(n) environment variable must be set. This environment variable determines palette behavior and must be set to enable 256-color support.

Part 3: Programming Tutorials

This part contains the following chapters:

17 Sample Programs

This chapter contains sample dialog for the following objects:

- Entry fields
- Push buttons
- Check boxes
- List boxes
- Scroll bars
- Tab controls

It also contains the following sample programs:

- Dsrnr
- Push-pop
- Custom1

17.1 Entry Fields

Entry fields are described in detail in the chapter *Control Objects*. This section describes dialog required to:

- Validate entry fields.
- Edit multiple line entry fields.
- Use a scroll bar with an entry field. See the section Events Associated with a Scroll Bar.

17.1.1 Validating Entry Fields

Entry fields are validated with the VALIDATE function. If validation fails, the VAL-ERROR event is triggered. You can validate an entry field explicitly:

```
VALIDATE AMOUNT-DEPOSITED-EF
```

Or you can validate an entry field implicitly by validating the parent window. For example:

```
VALIDATE DEPOSIT-WIN
```

The recommended way is to validate the field implicitly when the user accepts data on the window or dialog box that contains the field (for example the user presses **OK** or **Enter**). This lets the user enter the information, review it, and correct it if necessary before the validation. If the validation fails, you can set the focus back on the field with the error.

```
1 BUTTON-SELECTED
2     VALIDATE $WINDOW
3     INVOKE-MESSAGE-BOX ERROR-MB "All fields OK" $REGISTER
4     RETC
5 VAL-ERROR
6     INVOKE-MESSAGE-BOX ERROR-MB ERROR-MSG-FIELD $REGISTER
7     SET-FOCUS $EVENT-DATA
```

Line 1:

```
BUTTON-SELECTED
```

In this example, validation is activated by selecting a push button. Validation could also be activated by the window losing focus or a particular field losing focus using the LOST-FOCUS event. For example, focus is lost by the current control when the user uses the **Tab** key to move to another control.

Line 2:

```
    VALIDATE $WINDOW
```

This statement validates all fields on the current window according to the validation criteria you set up. If Dialog System detects an error, the VAL-ERROR event is triggered. If no error is detected, Dialog System continues as normal and executes the next function.

Lines 3-4:

```
     INVOKE-MESSAGE-BOX ERROR-MB "All fields OK" $REGISTER
     RETC
```

The VAL-ERROR event did not occur. Invoke a message box informing the user that no errors were detected and return to the program.

Line 5:

```
 VAL-ERROR
```

The VAL-ERROR event is detected. The special register $EVENT-DATA identifies the entry field that failed validation.

Line 6:

```
     INVOKE-MESSAGE-BOX ERROR-MB ERROR-MSG-FIELD $REGISTER
```

Notify the user of the error using a message box. The second parameter, `ERROR-MSG-FIELD`, contains the error message defined for this validation error. `ERROR-MSG-FIELD` is defined as the error message field in the data definition.

Line 7:

The following dialog illustrates one way of coding the validation step. The dialog is attached to a push button.

```
     SET-FOCUS $EVENT-DATA
```

Put the input focus back on the entry field that failed the validation.

Warning: You can validate the field when the user tries to move off it by using the LOST-FOCUS event . However, if the validation fails and you set the focus on the field, the focus always returns to the field whenever the user tries to put the focus on another application. It is better to validate the whole window.

17.1.1.1 Complex Data Validation

The procedures described in the previous section handle simple data validation. Sometimes, more complex validation than Dialog System can provide is needed. For example, the range of a valid credit limit often depends on the number of years a customer has been with a company.

This type of cross-field validation requires intervention because either you need data that is not contained in your screenset, or the validation is more sophisticated than the validation rules provided by Dialog

System. You need to return to the calling program to do this type of checking.

See the chapter *Advanced Topics* for an example.

17.1.2 Editing Multiple Line Entry Fields

You can move text to an associated data item using either your application program or dialog.

17.1.2.1 Moving Text Using Your Application Program

An example of moving text to an associated data item in your application program is the statement:

```
move "Now is the time..." to large-entry-field
```

This moves the string to the Data Block item that is associated with the MLE.

You can insert line feeds by using the hexadecimal character "0a". For example:

```
move "line1" & x"0a" & "line2"
```

where:

`line1` and `line2`	are the text to be displayed on separate lines.
`&`	concatenates the character strings.
`x`	indicates a hexadecimal character.
`0a`	is the hexadecimal character for a line feed.

17.1.2.2 Moving Text Using Dialog

In this case, line feeds cannot be inserted. For example, use the following statement in dialog:

```
move "Now is the time..." large-entry-field
```

where `large-entry-field` is the data item associated with the MLE.

17.2 Push Buttons

This section covers samples of dialog for:

- A **Pause** push button.
- Dynamically changing bitmaps associated with a push button.

17.2.1 Dialog for a Pause Push Button

```
1 BUTTON-SELECTED
2     DISABLE-OBJECT $CONTROL
3     ENABLE-OBJECT CONTINUE-PB
4     BRANCH-TO-PROCEDURE PAUSE-SELECTED
```

Line 1:

```
BUTTON-SELECTED
```

The user selects the **Pause** button, which triggers the BUTTON-SELECTED event. This event is the primary event associated with a push button.

Line 2:

```
DISABLE-OBJECT $CONTROL
```

`$CONTROL` is a special register that identifies the currently selected control. This function disables the current control (in this case, the **Pause** push button). This means the button is unavailable to the user.

Line 3:

```
ENABLE-OBJECT CONTINUE-PB
```

This statement enables the **Continue** push button. This means this button becomes available for the user to select. `CONTINUE-PB` is the name of the push button that you assigned to the push button in the Push Button properties window.

Line 4:

```
BRANCH-TO-PROCEDURE PAUSE-SELECTED
```

The functions in the PAUSE-SELECTED procedure are performed. In this procedure you can, for example, set a flag and return to the calling program.

For information on defining a push button, see the chapter *Control Objects*.

17.2.2 Dialog for Dynamically Changing Bitmaps Assigned to a Push Button

The following fragment of dialog illustrates one way to change dynamically the bitmaps assigned to a push button.

Note: Before you can use bitmaps, you must make them available to Dialog System. See the chapter *Control Objects* for further information.

```
1 ...
2 @SAVE
3     BRANCH-TO-PROCEDURE SAVE-FUNCTION
4   SAVE-FUNCTION
5     SET-OBJECT-LABEL GENERIC-PB SAVE-STATES
6     SET-FOCUS GENERIC-WIN
7     ...
```

Lines 2-3:

```
@SAVE
    BRANCH-TO-PROCEDURE SAVE-FUNCTION
```

The user selected the **Save** option from a menu. Branch to the procedure that does the save function.

Line 5:

```
    SET-OBJECT-LABEL GENERIC-PB SAVE-STATES
```

`GENERIC-PB` is the name assigned to the push button. `SAVE-STATES` is an alphanumeric data item large enough to hold the names of the replacement bitmaps and three null characters (x"`0A`").

Line 6:

```
    SET-FOCUS GENERIC-WIN
```

Set the keyboard focus to the window, so that all objects in the window are refreshed, and continue.

In your program, fill `save-states` in the Data Block using a statement like:

```
     move "save-normal" & x"0A" & "save-disabled" & x"0A" &
                   "save-depressed" & x"0A"  to save-states.
```

where `save-normal`, `save-disabled`, and `save-depressed` are the names of the bitmaps that are displayed depending on the state of the push button. The order you assign the names is important.

The first name is the bitmap that is displayed when the push button is in the "normal" state. The second name is the bitmap displayed when the push button is disabled. The third name is the bitmap displayed when the push button is pressed.

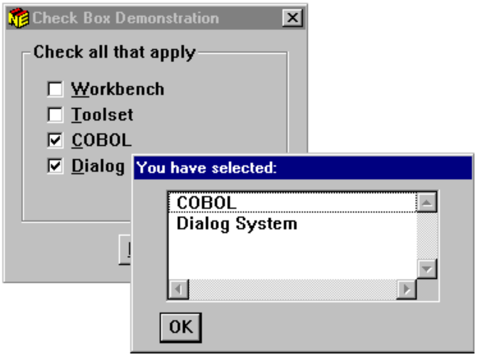
17.3 Check Boxes

Check boxes are described in detail in the chapter *Control Objects*. The **Objects** sample program shows an example of using check boxes.

17.3.1 Selecting Items From a List

This example shows you how to select one or more of the products from a list and display the items in a list box by clicking a **Display** push button. See Figure 17-1.

Figure 17-1. Check Boxes With List Box



Check Box Demonstration

Check all that apply

□ Workbench
□ Toolset
☑ COBOL
☑ Dialog

You have selected:

COBOL
Dialog System

The data definitions for this sample screenset are:

```
1   WORKBENCH                       9         1.00
2   TOOLSET                         9         1.00
3   COBOL                           9         1.00
4   DIALOG-SYSTEM                   9         1.00
```

```
5   PRODUCTS                                 4
6     PRODUCT-DISPLAY                    X        15.00
```

Data items 1 to 4 are the data items associated with the corresponding check boxes. Data items 5 and 6 are used for the list box that displays the selected items.

The dialog (attached to the **Display** push button) for controlling this example is:

```
 1 BUTTON-SELECTED
 2     MOVE "               " PRODUCT-DISPLAY(1)
 3     MOVE "               " PRODUCT-DISPLAY(2)
 4     MOVE "               " PRODUCT-DISPLAY(3)
 5     MOVE "               " PRODUCT-DISPLAY(4)
 6     MOVE 1 $REGISTER
 7     IF= WORKBENCH 0 CHECK-TOOLSET
 8     MOVE "Workbench" PRODUCT-DISPLAY($REGISTER)
 9     INCREMENT $REGISTER
10     BRANCH-TO-PROCEDURE CHECK-TOOLSET
11
12   CHECK-TOOLSET
13     IF= TOOLSET 0 CHECK-COBOL
14     MOVE "Toolset" PRODUCT-DISPLAY($REGISTER)
15     INCREMENT $REGISTER
16     BRANCH-TO-PROCEDURE CHECK-COBOL
17
18   CHECK-COBOL
19     IF= COBOL 0 CHECK-DIALOG-SYSTEM
20     MOVE "COBOL" PRODUCT-DISPLAY($REGISTER)
21     INCREMENT $REGISTER
22     BRANCH-TO-PROCEDURE CHECK-DIALOG-SYSTEM
23
24   CHECK-DIALOG-SYSTEM
25     IF= DIALOG-SYSTEM 0 DISPLAY-PRODUCTS-DB
26     MOVE "Dialog System" PRODUCT-DISPLAY($REGISTER)
27     BRANCH-TO-PROCEDURE DISPLAY-PRODUCTS-DB
28
29   DISPLAY-PRODUCTS-DB
30     REFRESH-OBJECT CHECKB-LB
31     SET-FOCUS CHECKB-DB
```

Line 1:

```
BUTTON-SELECTED
```

The user selected the **Display** push button .

Lines 2-5:

```
  MOVE "               " PRODUCT-DISPLAY(1)
  MOVE "               " PRODUCT-DISPLAY(2)
  MOVE "               " PRODUCT-DISPLAY(3)
  MOVE "               " PRODUCT-DISPLAY(4)
```

Initialize the list box items, by moving a blank string to the items in the PRODUCT-DISPLAY group.

Line 6:

```
  MOVE 1 $REGISTER
```

Initialize the index ($REGISTER) . $REGISTER is an internal register that you can use to store a numeric value. In this case, use it as an index into the list box.

Line 7:

```
  IF= WORKBENCH 0 CHECK-TOOLSET
```

If the Workbench check box has not been checked, branch to the procedure that checks if the Toolset check box has been checked. If the Workbench check box has been checked, continue to the next function and do not branch.

Lines 8-9:

```
  MOVE "Workbench" PRODUCT-DISPLAY($REGISTER)
  INCREMENT $REGISTER
```

To get here, the user selected the Workbench check box. Fill the list box data item and increment the index.

Line 10:

```
  BRANCH-TO-PROCEDURE CHECK-TOOLSET
```

Branch to the procedure that checks the Toolset check box.

Lines 12-27:

```
CHECK-TOOLSET
  IF= TOOLSET 0 CHECK-COBOL
  MOVE "Toolset" PRODUCT-DISPLAY($REGISTER)
  INCREMENT $REGISTER
  BRANCH-TO-PROCEDURE CHECK-COBOL

CHECK-COBOL
  IF= COBOL 0 CHECK-DIALOG-SYSTEM
  MOVE "COBOL" PRODUCT-DISPLAY($REGISTER)
  INCREMENT $REGISTER
  BRANCH-TO-PROCEDURE CHECK-DIALOG-SYSTEM

CHECK-DIALOG-SYSTEM
  IF= DIALOG-SYSTEM 0 DISPLAY-PRODUCTS-DB
  MOVE "Dialog System" PRODUCT-DISPLAY($REGISTER)
```

```
     BRANCH-TO-PROCEDURE DISPLAY-PRODUCTS-DB
```

This is similar dialog for the other check boxes.

Lines 29-31:

```
   DISPLAY-PRODUCTS-DB
     REFRESH-OBJECT CHECKB-LB
     SET-FOCUS CHECKB-DB
```

Display the selected products in the list box.

17.4 List Boxes

You can update items in a list box in three ways:

- By using a group item.
- By using dialog which updates the items at run time.
- By using a delimited string.

17.4.1 Adding Items Using Group Item

You can associate the list box with a group item, so that each line in the list box displays an occurrence of the group. Selected items in the group then occupy fields in each line.

The sample application **Saledata** illustrates this method of accessing a list box. Using an existing file, the application loads a set of sales data to a group item (SALES-GROUP) in the Data Block. The list box (SALES-LB) displays the data. Then you can update (insert, change, delete) entries or view the set of data in different sort orders.

This Data Block section defines the data referenced in the following dialog:

```
SALES-GROUP              100
  SALES-NAME       X        20.00
  SALES-REGION     X         4.00
  SALES-STATE      X         2.00
TMP-NAME           X        20.00
TMP-REGION         X         4.00
```

```
TMP-STATE          X          2.00
NUMBER-OF-RECORDS  9          3.00
```

The following portions of dialog are those dealing with the list box:

```
 1     SET-DATA-GROUP-SIZE SALES-GROUP NUMBER-OF-RECORDS
 2 ...
 3 ITEM-SELECTED
 4     MOVE $EVENT-DATA $REGISTER
 5 ...
 6   PROC-INSERT
 7     INSERT-OCCURRENCE SALES-GROUP $REGISTER
 8     MOVE TMP-NAME SALES-NAME($REGISTER)
 9     MOVE TMP-REGION SALES-REGION($REGISTER)
10     MOVE TMP-STATE SALES-STATE($REGISTER)
11     INCREMENT NUMBER-OF-RECORDS
12     INCREMENT $REGISTER
13     REFRESH-OBJECT SALES-LB
14     SET-LIST-ITEM-STATE SALES-LB 1 $REGISTER
15 ...
16   PROC-CHANGE
17     MOVE TMP-NAME SALES-NAME($REGISTER)
18     MOVE TMP-REGION SALES-REGION($REGISTER)
19     MOVE TMP-STATE SALES-STATE($REGISTER)
20     UPDATE-LIST-ITEM SALES-LB SALES-GROUP $REGISTER
21     SET-LIST-ITEM-STATE SALES-LB 1 $REGISTER
22 ...
23   PROC-DELETE
24     DELETE-OCCURRENCE SALES-GROUP $REGISTER
25     DECREMENT NUMBER-OF-RECORDS
26     DECREMENT $REGISTER
27     REFRESH-OBJECT SALES-LB
28     SET-LIST-ITEM SALES-LB 1 $REGISTER
```

Line 1:

```
    SET-DATA-GROUP-SIZE SALES-GROUP NUMBER-OF-RECORDS
```

This statement defines the internal size of the data group. This is the number of occurrences of a group that you can access. Refer to the description of the SET-DATA-GROUP-SIZE function in your help for a complete definition of internal size.

Lines 3-4:

```
 ITEM-SELECTED
     MOVE $EVENT-DATA $REGISTER
```

You can browse the list box until you find the item you want to select. The ITEM-SELECTED event is triggered when the item (list box row) is selected. The special event register $EVENT-DATA contains the row number of the item selected. Use $REGISTER to keep track of the current location in both the group data and the list box.

Line 6:

```
  PROC-INSERT
```

This procedure inserts an item in the position now occupied by the selected item.

Line 7:

```
    INSERT-OCCURRENCE SALES-GROUP $REGISTER
```

Insert a blank occurrence into the data group at the specified location. The selected item and all items following are moved down one row. The list box is not affected. This function only updates the data group.

Lines 8-10:

```
    MOVE TMP-NAME SALES-NAME($REGISTER)
    MOVE TMP-REGION SALES-REGION($REGISTER)
    MOVE TMP-STATE SALES-STATE($REGISTER)
```

Update the group item with the new information.

Line 11:

```
    INCREMENT NUMBER-OF-RECORDS
```

Increment the number of records in the data group.

Line 12:

```
    INCREMENT $REGISTER
```

Update the pointer to the current row. This keeps the pointer addressing the same row it was before the insertion.

Line 13:

```
    REFRESH-OBJECT SALES-LB
```

A list box associated with a group item must be refreshed before it reflects changes in the data. The INSERT-OCCURRENCE and DELETE-OCCURRENCE functions have no effect on a list box. If you have only one entry in the list box that needs to be refreshed, you can use the UPDATE-LIST-ITEM function. However, in the case of an insertion, all rows in the data group after the insertion row are changed. It is better to use the REFRESH-OBJECT function.

Line 14:

```
    SET-LIST-ITEM-STATE SALES-LB 1 $REGISTER
```

Change the state of the current row to "selected" . This is the same row that was selected before the insertion.

Line 16:

```
  PROC-CHANGE
```

This procedure changes the contents of the selected item.

Lines 17-19:

```
      MOVE TMP-NAME SALES-NAME($REGISTER)
      MOVE TMP-REGION SALES-REGION($REGISTER)
      MOVE TMP-STATE SALES-STATE($REGISTER)
```

Update the group item with the new information.

Line 20:

```
      UPDATE-LIST-ITEM SALES-LB SALES-GROUP $REGISTER
```

Because only one row in the list box changed, you can use the UPDATE-LIST-ITEM function. You do not have to refresh the entire list box since all other rows remain the same.

Line 21:

```
      SET-LIST-ITEM-STATE SALES-LB 1 $REGISTER
```

Change the state of the current row to "selected". This is the same row that was selected before the change.

Line 24:

```
      DELETE-OCCURRENCE SALES-GROUP $REGISTER
```

Delete the occurrence from the data group at the selected location. All items following are moved up one row.

Line 25:

```
      DECREMENT NUMBER-OF-RECORDS
```

Decrement the number of records in the data group.

Line 26:

```
      DECREMENT $REGISTER
```

Update the pointer to the current row. This keeps the pointer addressing the same row.

Line 27:

```
      REFRESH-OBJECT SALES-LB
```

Refresh the list box.

Line 28:

```
      SET-LIST-ITEM SALES-LB 1 $REGISTER
```

Change the state of the current row to "selected". This is the row following the one that was deleted.

17.4.2 Adding Items Using Dialog

You can also maintain items in a list box at run time using the dialog functions INSERT-LIST-ITEM, UPDATE-LIST-ITEM, and DELETE-LIST-ITEM .

For example, the following dialog fragment is one way to add the abbreviations for months to a list box named MONTH-LB.

```
SCREENSET-INITIALIZED
    INSERT-LIST-ITEM MONTH-LB "Jan" 1
    INSERT-LIST-ITEM MONTH-LB "Feb" 2
    INSERT-LIST-ITEM MONTH-LB "Mar" 3
    INSERT-LIST-ITEM MONTH-LB "Apr" 4
    INSERT-LIST-ITEM MONTH-LB "May" 5
    INSERT-LIST-ITEM MONTH-LB "Jun" 6
    INSERT-LIST-ITEM MONTH-LB "Jul" 7
    INSERT-LIST-ITEM MONTH-LB "Aug" 8
    INSERT-LIST-ITEM MONTH-LB "Sep" 9
    INSERT-LIST-ITEM MONTH-LB "Oct" 10
    INSERT-LIST-ITEM MONTH-LB "Nov" 11
    INSERT-LIST-ITEM MONTH-LB "Dec" 12
```

This is an effective way of populating a list box if you have a few choices in your list and you want to keep the Data Block as small as possible. Although it does not require much time to fill a small list box this way, it does take a little time and does add to the number of dialog statements that have to be searched. The section *How Dialog System Searches for Event Dialog* in the chapter *Using Dialog* describes the rules for searching dialog.

As an alternative, the following dialog can be used:

```
SCREENSET-INITIALIZED
    INSERT-LIST-ITEM MONTH-LB "Jan" 0
    INSERT-LIST-ITEM MONTH-LB "Feb" 0
    INSERT-LIST-ITEM MONTH-LB "Mar" 0
    INSERT-LIST-ITEM MONTH-LB "Apr" 0
    INSERT-LIST-ITEM MONTH-LB "May" 0
    INSERT-LIST-ITEM MONTH-LB "Jun" 0
    INSERT-LIST-ITEM MONTH-LB "Jul" 0
    INSERT-LIST-ITEM MONTH-LB "Aug" 0
    INSERT-LIST-ITEM MONTH-LB "Sep" 0
    INSERT-LIST-ITEM MONTH-LB "Oct" 0
    INSERT-LIST-ITEM MONTH-LB "Nov" 0
    INSERT-LIST-ITEM MONTH-LB "Dec" 0
```

A value of 0 for the third parameter tells Dialog System to insert the data item at the end of the list.

17.4.3 Adding Items Using a Delimited String

Another way to insert a few items in a list box is the INSERT-MANY-LIST-ITEMS function using a data item passed from your program. For example, the following dialog inserts the same month abbreviations into MONTH-LB:

```
SCREENSET-INITIALIZED
    INSERT-MANY-LIST-ITEMS MONTH-LB MONTHS-STRING 48
```

where the parameters are:

MONTH-LB	The name of the list box you defined in the List Box Properties dialog box.
MONTHS-STRING	A data item, passed from your program, with the individual list items delimited by x"0A". It is defined in the Data Definition facility as: MONTHS-STRING X 48
48	The number of characters to copy to the list box. See the topic *INSERT-MANY-LIST-ITEMS* in the Help for more details about this parameter.

In the Working-Storage Section of your program, define the string as:

```
01 months.
    03 pic x(4) value "Jan" & x"0A".
    03 pic x(4) value "Feb" & x"0A".
    03 pic x(4) value "Mar" & x"0A".
    03 pic x(4) value "Apr" & x"0A".
    03 pic x(4) value "May" & x"0A".
    03 pic x(4) value "Jun" & x"0A".
    03 pic x(4) value "Jul" & x"0A".
    03 pic x(4) value "Aug" & x"0A".
    03 pic x(4) value "Sep" & x"0A".
    03 pic x(4) value "Oct" & x"0A".
    03 pic x(4) value "Nov" & x"0A".
    03 pic x(4) value "Dec" & x"0A".
```

Then move the string to the Data Block with a statement like:

```
move months to months-string
```

17.5 Scroll Bars

This section covers:

- Events associated with a scroll bar.
- Scroll bar properties.

17.5.1 Events Associated with a Scroll Bar

Two events are associated with a scroll bar; SLIDER-MOVING and SLIDER-RELEASED.

The SLIDER-MOVING event occurs when the slider has been moved to a new position.

For example, in the dialog to implement this feature:

```
1 SLIDER-MOVING
2     MOVE $EVENT-DATA COUNTER
3     REFRESH-OBJECT COUNTER-DISP
4 SLIDER-RELEASED
5     MOVE $EVENT-DATA COUNTER
6     REFRESH-OBJECT COUNTER-DISP
```

lines 1 to 3 display the position of the slider in the entry field `COUNTER-DISP`.

The SLIDER-RELEASED event occurs when the slider has been released. Lines 4 to 6 in the dialog use this event and also display the position of the slider. A complete description of this dialog is:

Line 1:

```
     SLIDER-MOVING
```

The slider is moved to a new position. This triggers the SLIDER-MOVING event.

Line 2:

```
     MOVE $EVENT-DATA COUNTER
```

When the slider is moved, the special register $EVENT-DATA contains the new slider position. This position is expressed in relation to the minimum and maximum values. COUNTER is a Data Block data item that is attached to the entry field COUNTER-DISP.

Line 3:

```
     REFRESH-OBJECT COUNTER-DISP
```

The entry field must be refreshed to reflect the new data value.

Line 4:

```
     SLIDER-RELEASED
```

The slider is released at a new position. This triggers the SLIDER-RELEASED event.

Line 5:

```
     MOVE $EVENT-DATA COUNTER
```

When the slider is released, $EVENT-DATA contains the new slider position. Again, COUNTER is a Data Block data item that is attached to the entry field COUNTER-DISP.

Line 6:

```
     REFRESH-OBJECT COUNTER-DISP
```

Refresh the entry field.

The scroll bar example in the sample application **Objects** shows how to use an entry field this way.

17.5.2 Scroll Bar Properties

The Scroll Bar Properties dialog box enables you to assign defaults to scroll bar properties such as slider range, slider position and slider size. Using dialog, you can change these properties .

For example, this dialog fragment changes the properties of a scroll bar named EMP-LIST-SB.

```
SET-SLIDER-RANGE EMP-LIST-SB 0 50
SET-SLIDER-POSITION EMP-LIST-SB 25
SET-SLIDER-SIZE EMP-LIST-SB 5
```

Refer to the topic *Dialog Statements: Functions* in the Help for a full description of these functions.

For information on defining a scroll bar, see the topic *Objects and Properties* in the Help.

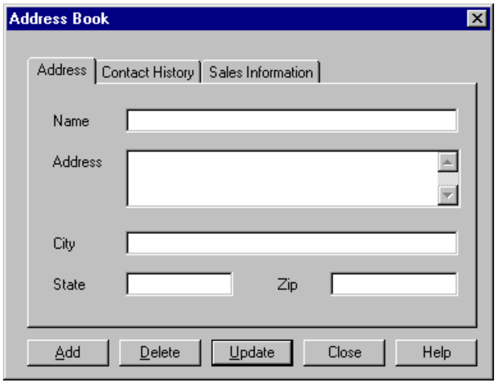
17.6 Tab Controls

Pages can be inserted and deleted from a tab control using the COPY-PAGE and DELETE-PAGE functions. This enables the tab control to be maintained dynamically.

As an example, you could write an application which stored customer contact information such as address, contact history, and sales information. This information for each customer could be displayed in a tab control.

See Figure 17-2 below.

Figure 17-2. Sample Tab Control Page



If, for example, one page of sales information was not sufficient, you could add a second page as follows:

```
1    ADD-MORE-SALES-INFO
2      COPY-PAGE SALES-INFO-PAGE 0 2
3      SET-OBJECT-LABEL SALES-INFO-PAGE
                                     "Sales Information 2"
```

Line 1:

```
ADD-MORE-SALES-INFO
```

A procedure to add another sales information page to the tab control.

Line 2:

```
    COPY-PAGE SALES-INFO-PAGE 0 2
```

COPY-PAGE makes a copy of the tab control page and inserts it in the correct location in the tab control. `SALES-INFO-PAGE` is the name assigned to the page. The second parameter, `0`, is the position to copy to; 0 means last, 1 means first. The third parameter specifies the instance number for the page. The instance number overrides the subscript of any entry fields with subscripted master fields that appear on the page. Remember, however, that any pages you create in the tab control will always appear unless you explicitly delete them (see below).

Line 3:

```
    SET-OBJECT-LABEL SALES-INFO-PAGE "Sales Information 2"
```

Changes the text in the tab on the new page. Note that after a COPY-PAGE, `SALES-INFO-PAGE` refers to the new page, not the original.

To delete a page from a tab control, the DELETE-PAGE function is used. For example, if a customer had no sales information, you could delete the page as follows:

```
BUTTON-SELECTED
    DELETE-PAGE SALES-INFO-PAGE
```

See the Help for a detailed description of these functions.

17.7 The Call Interface

Details about using the call interface can be found in the chapter *Using the Screenset*. This section covers:

- Using Dsrnr.
- Pushing and popping screensets.
- Using multiple instances of screensets.

17.7.1 Using Dsrnr

Dsrnr is a sample subprogram supplied with Dialog System. To launch it:

1 Start Dsrunner (see the chapter *Multiple Screensets* for full details).

runw dsrunner

2 Select **Open a screenset**on the **File** menu in the Dsrunner window.

3 Select **Dsrnr** from the file selector that is displayed (it is in your demonstration directory).

The Dsrnr main window appears. You can load multiple instances of Dsrnr (you can have up to 32 screensets on the stack at any time) simply by opening it again. If you want, you can run the subprogram through the Animator.

Now let’s take a detailed look at key sections of the sample code.

```
 linkage section.

* data block from screenset
 copy "dsrnr.cpb".

* (optional) dsrunner info and ds event blocks
 copy "dsrunner.cpy".
 copy "dssysinf.cpy".
```

You must copy the Data Block copyfile into your program to be able to access the screenset data. You can, optionally, also copy in the dsrunner copyfile that represents the linkage between Dsrunner and your subprogram, dsrunner.cpy. If you want to process Panels2 event information you also need to copy in **dssysinf.cpy**. Notice that these are copied into the linkage section because they are parameters of the subprogram.

```
 procedure division using data-block
                          dsrunner-info-block
                          ds-event-block.
```

You must specify the data-block parameter. The other two parameters are optional. Use dsrunner-info-block if you want to be informed of errors. Use ds-event-block if you want to use Panels2 and Dialog System together.

```
 main section.
* Put out a message if we are run from the command line
* instead of being called by DSRUNNER.
     if (address of data-block = null)
        display "This is a subprogram, and must be CALLed"
*       display "(Preferably from DSRUNNER)."
        exit program
        stop run
     end-if
```

You need to check that the subprogram is being called and you can do that easily by checking that it is being passed a Data Block address.

```
     move 0 to return-code

* Determine if this is the first time we’ve been called
* with this data block. We can do this by checking the
* data blockfor low values, which is what DSRUNNER
* initialises all data blocks with.
     if (data-block = all low-values)
* This is the first time we have been called with this
* Screenset/Data-Block. Perform any initialisation,
* and then exit. The screenset is not loaded (and no
* SCREENSET-INITIALIZED event occurs) until after
* we return from here.
         perform initialisation
     else
* This is not the first time we have been called,
* therefore do our normal handling.
         perform handle-screenset-request
     end-if

     continue.

```

```
58 exit-main.
59
60     exit program
61     stop run.
```

Each time the subprogram is called as a new instance, it must perform any required initialization of the Data Block. Line 43 shows how to check if this is the first time the subprogram has been called with this Data Block (new instance of screenset). Once the subprogram has performed its initialization, you must return to Dsrunner. Dsrunner will call your program whenever a RETC occurs in the screenset associated with this subprogram.

```
64 initialisation section.
65* Perform any pre-screenset initialisation here.
66
67     initialize data-block
68     move dsrunner-screenset-instance to my-instance-no
69
70 continue.
```

Line 68 shows how the subprogram can find out its instance number.

```
73 handle-screenset-request section.
74* We end up here whenever the screenset does a RETC, or
75* a DSGRUN error has occurred, and DSRUNNER is calling us
76* to do any error handling before we get terminated.
77
78* Have we been called because screenset caused an error?
79     if (dsrunner-error-code not = 0)
80* Yes, so handle it
81         perform handle-dsgrun-error
82         exit section
83     end-if
84* Have we been called because of a validation error?
85     if (dsrunner-validation-error-no not = 0)
86* Yes, so handle it
87         perform handle-validation-error
88         exit section
89     end-if
90
91* Must have been a normal RETC from our screenset, so
92* just service the screenset request.
93
94     move "successful" to program-string
95
```

```
 96     evaluate reason-for-returning
 97      when "+"
 98         add program-value-1 to program-value-2
 99                             giving result-value
100
101      when "-"
102         subtract program-value-1 from program-value-2
103                                  giving result-value
104
105      when "*"
106         multiply program-value-1 by program-value-2
107                                  giving result-value
108
109      when "/"
110         divide program-value-1 by program-value-2
111                                giving result-value
112          on size error
113             move "Bad result from divide"
114                                        to program-string
115      when other
116         move "sorry, unsupported function"
117                                        to program-string
118     end-evaluate
119
120     continue.
```

This section of code handles screenset requests following a RETC dialog function within the screenset associated with this subprogram.

Checks are made in line 79 and in line 85 to see if there has been a Dsrunner error or a validation error of some sort. If either error has occurred it is handled accordingly, otherwise the screenset request is handled.

```
123 handle-dsgrun-error section.
124* We end up here when a DSGRUN error occurs in the
125* screenset. In this example I am deciding to display the
126* error message myself. If this is the first time this
127* has occurred, I continue, otherwise I set the
128* RETURN-CODE indicating that DSRUNNER should terminate
129* me. If the return-code is zero DSRUNNER would continue
130* as if nothing had happened.
131     move dsrunner-error-code to error-number
132     move dsrunner-error-details-1 to error-details1
133     move dsrunner-error-details-2 to error-details2
134     display "dsrnr: "
135             "dsgrun error " error-number
136             ", " error-details1
```

```
137             ", " error-details2
138
139     if (handle-error-count = 0)
140         add 1 to handle-error-count
141     else
142* This will force dsrunner to terminate me
143         move 1 to return-code
144     end-if
145
146     continue.
```

This section of code handles dsrunner errors.

Line 143 shows how to terminate Dsrunner by returning a non-zero return code.

```
149 handle-validation-error section.
150* We end up here when a screenset validation error has
151* occurred. Same rules apply here as for above.
152
153     move dsrunner-validation-error-no to error-number
154
155     display "dsrnr: validation error code " error-number
156
157* This will force dsrunner to terminate me
158     move 1 to return-code
159
160     continue.
```

This section of code handles user input errors.

17.7.2 The Push-pop Sample Program

The following sample program, **push-pop.cbl**, demonstrates the use of the pushing and popping of screensets.

The COBOL programs, screensets, and associated files are included on your samples disk. Instructions for compiling and running these programs are given in the chapter *Using the Screenset*.

Push-pop uses three screensets:

- A file manager screenset.
- A print manager screenset.

- A main screen from which the other screensets are called.

A full listing of the program is provided below. An explanation of the functions used for pushing and popping is given after.

```
$SET ANS85 MF

working-storage section.
    copy "ds-cntrl.mf  ".
    copy "pushmain.cpb ".
    copy "filemgr.cpb  ".
    copy "printmgr.cpb ".

01 new-screenset-name       pic x(12).

01 action                   pic 9.
   78 load-file                        value 1.
   78 load-print                       value 2.
   78 exit-program                     value 3.
01 end-of-actions-flag      pic 9.
   88 end-of-actions                   value 1.

procedure division.

main-process.
    perform program-initialize
     call "dsgrun" using ds-control-block,
                         pushmain-data-block
    perform process-actions until end-of-actions
stop run.

program-initialize.
    initialize ds-control-block
    initialize pushmain-data-block
    move ds-new-set to  ds-control
    move pushmain-data-block-version-no to
       ds-data-block-version-no
    move pushmain-version-no to ds-version-no
    move "pushmain" to ds-set-name
    move zero to end-of-actions-flag.

process-actions.
    evaluate true
      when pushmain-action = load-file
        move "filemgr" to ds-set-name
        move ds-push-set to ds-control
        move 1 to ds-control-param
        initialize filemgr-data-block
```

```
44           move filemgr-data-block-version-no to
45             ds-data-block-version-no
46           move filemgr-version-no to ds-version-no
47           call "dsgrun" using ds-control-block,
48                                   filemgr-data-block
49           perform file-mgr-work
50
51         when pushmain-action = load-print
52           move "printmgr" to ds-set-name
53           move ds-push-set to ds-control
54           move 1 to ds-control-param
55           initialize printmgr-data-block
56           move printmgr-data-block-version-no to
57             ds-data-block-version-no
58           move printmgr-version-no to ds-version-no
59           call "dsgrun" using ds-control-block,
60                                  printmgr-data-block
61           perform print-mgr-work
62           when pushmain-action = exit-program
63           move 1 to end-of-actions-flag
64       end-evaluate.
65
66   file-mgr-work.
67           move ds-quit-set to ds-control
68           call "dsgrun" using ds-control-block,
69             filemgr-data-block
70
71           move ds-continue to ds-control
72           call "dsgrun" using ds-control-block,
73                                   pushmain-data-block.
74
75   print-mgr-work.
76           move ds-quit-set to ds-control
77           call "dsgrun" using ds-control-block,
78             printmgr-data-block
79           move ds-continue to ds-control
80           call "dsgrun" using ds-control-block,
81                                   pushmain-data-block.
```

The function of this code is detailed as follows:

Lines 1-7:

```
1  $SET ANS85 MF
2
3  working-storage section.
4      copy "ds-cntrl.mf  ".
5      copy "pushmain.cpb ".
6      copy "filemgr.cpb  ".
7      copy "printmgr.cpb ".
```

The first section of the program copies the appropriate Control Block copyfile, and the generated copyfiles for each screenset that will be used.

Line 9:

```
9  01 new-screenset-name    pic x(12).
```

Next, the picture string for `new-screenset-name` is defined.

Lines 11-16:

```
11   01 action                    pic 9.
12      78 load-file                          value 1.
13      78 load-print                         value 2.
14      78 exit-program                       value 3.
15   01 end-of-actions-flag      pic 9.
16      88 end-of-actions                     value 1.
```

Then, the values for the actions that will cause a particular screenset to be loaded are declared.

Lines 20-25:

```
20   main-process.
21      perform program-initialize
22      call "dsgrun" using ds-control-block,
23                          pushmain-data-block
24      perform process-actions until end-of-actions
25      stop run.
```

The first procedure in the program is the `main-process`. This performs the routine `program-initialization`, then calls Dsgrun using `ds-control-block` and the Data Block for the `pushmain` screenset. It performs `process-actions` until `end-of-actions` is received and `stop-run` occurs.

Lines 27-35:

```
27 program-initialize.
28      initialize ds-control-block
29      initialize pushmain-data-block
30      move ds-new-set to ds-control
31      move pushmain-data-block-version-no to
32        ds-data-block-version-no
33      move pushmain-version-no to ds-version-no
34      move "pushmain" to ds-set-name
35      move zero to end-of-actions-flag.
```

The `program-initialize` procedure initializes the Control and Data Blocks, and moves the appropriate values for the screenset `pushmain`. Note that line 30 places the value **N** in `ds-control`. This is the value you should use if Dsgrun has not yet been called and started a screenset.

Lines 37-39:

```
37   process-actions.
38      evaluate true
39        when pushmain-action = load-file
```

This section of the program performs evaluations to test whether a new screenset should be loaded. The main screenset is now loaded and has focus until the value of `pushmain-action` changes. If the value of `pushmain-action` becomes `load-file`, the screenset for the file manager will be loaded.

Lines 40-41:

```
40   move "filemgr" to ds-set-name
41   move ds-push-set to ds-control
```

When `pushmain-action` is equal to `load-file`:

- The screenset name, `filemgr`, is moved to `ds-set-name`
- The value `ds-push-set` is moved to `ds-control`.

Note: `ds-push-set` is a level-78 data item in `ds-cntrl.mf`, with a defined value of **S**.

Lines 43-49:

```
43   initialize filemgr-data-block
44   move filemgr-data-block-version-no to
45     ds-data-block-version-no
46   move filemgr-version-no to ds-version-no
47   call "dsgrun" using ds-control-block,
48                       filemgr-data-block
49   perform file-mgr-work
```

This section of the program performs initialization of the Control Block and Data Block, and checks version information. It then calls Dsgrun using the file manager screenset.

Lines 51-61:

```
51   when pushmain-action = load-print
52   move "printmgr" to ds-set-name
53   move ds-push-set to ds-control
54   move 1 to ds-control-param
55   initialize printmgr-data-block
56   move printmgr-data-block-version-no to
57     ds-data-block-version-no
58   move printmgr-version-no to ds-version-no
```

```
59   call "dsgrun" using ds-control-block,
60                         printmgr-data-block
61   perform print-mgr-work
```

This section performs the identical functions for the print manager screenset if pushmain-action becomes 2.

Lines 62-63:

```
62   when pushmain-action = exit-program
63   move 1 to end-of-actions-flag
```

When pushmain-action is equal to exit-program, the value 1 is moved to the end-of-actions-flag, which terminates the program.

Lines 66-69:

```
66   file-mgr-work.
67   move ds-quit-set to ds-control
68   call "dsgrun" using ds-control-block,
69                         filemgr-data-block
```

File management functions are performed by Dialog System and not the calling program. When they are completed, the active screenset is closed and the new screenset is popped off the stack. This is performed by moving ds-quit-set to ds-control. When Dsgrun is called it will use ds-control-block but ignore filemgr-data-block.

Lines 71-73:

```
71   move ds-continue to ds-control
72   call "dsgrun" using ds-control-block,
73                         pushmain-data-block.
```

The program then moves ds-continue to ds-control causing the screenset pushmain to be popped from the screenset stack. Processing is continued from where it left off.

Lines 75-81:

```
75   print-mgr-work.
76   move ds-quit-set to ds-control
77   call "dsgrun" using ds-control-block,
78                         printmgr-data-block
79   move ds-continue to ds-control
80   call "dsgrun" using ds-control-block,
81                         pushmain-data-block.
```

The same operation is repeated for the print management functions.

17.7.2.1 The Custom1 Sample Program

custom1.cbl demonstrates the use of multiple instances of the same screenset. Because it is a long program, only those sections relevant to the use of multiple instances are shown. The COBOL programs, screensets, and associated files are included on your samples disk.

The program uses a main screenset called Custom1 and multiple instances of a second screenset called Custom2, which is mapped onto the items in a data group.

Lines 45-46:

```
45   78 main-ss-name                  value "custom1".
46   78 instance-ss-name              value "custom2".
```

Sets up the picture strings for the main screenset name and the instance screenset name in the Working-Storage Section of your program.

Lines 48-51:

```
48   copy "ds-cntrl.mf  ".
49   copy "custom1.cpb  ".
50   copy "custom2.cpb  ".
51   copy "dssysinf.cpy ".
```

Copies the various copyfiles into the program working storage area. Note that **dssysinf.cpy** is copied. This copyfile is always required if you are using multiple instances.

Lines 53-54:

```
53   01 instance-table             value all x"00".
54      03 group-record-no         pic 9(2) comp-x occurs 32.
55   01 group-index                pic 9(2) comp-x value 0.
```

Sets up an instance table to map the data group onto the screenset fields.

Lines 57-61:

```
57   78  refresh-text-and-data-proc   value "p255".
58   78  dialog-system                value "dsgrun".
59
60     77  array-ind                pic 9(4) comp.
61     77  display-error-no         pic 9(4).
```

Declares various values. The dialog procedure p255 is used to refresh the text and data.

Lines 63-64:

```
63   01 main-screenset-id             pic x(8).
64   01 instance-screenset-id         pic x(8).
```

Defines picture strings for the `main-screenset-id` and `instance-screenset-id`.

Line 66:

```
66   01 temp-word                     pic 9(4) comp-x.
```

The `temp-word` picture string is used to hold the value indicating the active Data Block. In effect it is a function code. This is applied later in the program where the value of `temp-word` can be evaluated and the appropriate action performed.

Line 103:

```
103   move data-block-ptr(1:2) to temp-word(1:2)
```

A problem when multiple screensets are used is to know which screenset Data Block is active. This code moves the first two bytes of the Data Block into `temp-word` where it is used as a function code.

Lines 104-143:

```
104   evaluate temp-word
105
106   when 1
107     perform set-up-for-ss-change
108     move x"0000" to data-block-ptr(1:2)
109
110   when 2
111      perform poss-invoke-new-instance
112      move x"0000" to data-block-ptr(1:2)
113
114   when 3
115     perform close-instance
116     move x"0000" to data-block-ptr(1:2)
117
118   when 4
119     perform update-details
120     move x"0000" to data-block-ptr(1:2)
121
122   when 5
123     perform close-all-instances
124     move x"0000" to data-block-ptr(1:2)
...
141   end-evaluate
142     perform clear-flags
143     perform call-dialog-system.
```

Evaluates the value of `temp-word` and performs the appropriate action. When the evaluation finishes, all flags are cleared and Dialog System is called.

Lines 271-287:

```
271   call-dialog-system section.
272
273   call dialog-system using ds-control-block,
274                            data-block-ptr
275                            ds-event-block
276   if (ignore-error = 0)
277     if not ds-no-error
278       move ds-error-code to display-error-no
279       move ds-error-details-1 to display-error-details1
280       move ds-error-details-2 to display-error-details2
281       display "ds error no: display-error-no
282          ", " display-error-details1
283          ", " display-error-details2
284          "Screenset is " ds-set-name
285     end-if
286   end-if
287   .
```

Calls Dialog System using `ds-control-block`, `data-block-ptr` (the pointer to the particular screenset instance Data Block), and `ds-event-block`. If Dialog System cannot be called, it displays an error.

Lines 284-297:

```
294   set-up-for-ss-change section.
295     move ds-event-screenset-id to ds-set-name
296     move ds-use-instance-set to ds-control
297     move ds-event-screenset-instance-no to
298                     ds-screenset-instance
```

Moves the parameters to Dsgrun to cause a change to a new screenset as a result of an OTHER-SCREENSET event. You could check whether you want to go to the first screenset then move `ds-use-set` instead of `ds-use-instance-set`. The first screenset instance number is still returned, even though there is only one on the stack.

Lines 319-329:

```
319   poss-invoke-new-instance section.
320
321     set not-found to true
322     move 0 to group-index
323     perform until found or group-index = 10
324
325     add 1 to group-index
326     if group-record-no(group-index) =
327                  customer-index-of-interest
328       set found to true
329     end-if
```

Checks to see if the required group-occurrence has already been instantiated.

Lines 331-341:

```
331   if found
332     move ds-use-instance-set to ds-control
333     move instance-screenset-id to ds-set-name
334     move group-index to ds-screenset-instance
335     move group-record-no(ds-screenset-instance)
336                               to group-index
337
338     move "show-yourself" to ds-procedure
339     move customer-group-001-item(group-index) to
340                                  redef-block
341     set address of data-block-ptr
                                     to address of data-block
```

If the group occurrence has been found, brings it into focus.

Lines 344-348:

```
344     move ds-push-set to ds-control
345     move instance-ss-name to ds-set-name
346   move data-block-version-no to ds-data-block-version-no
347     move version-no to ds-version-no
348     move ds-screen-noclear to ds-control-param
```

If the group occurrence is not found, creates a new screenset occurrence.

Lines 353-361:

```
353     move 1 to ds-clear-dialog
354     move "init-proc" to s-procedure
355
356     move customer-index-of-interest to group-index
357     move customer-group-001-item(group-index) to
358                                  redef-block
359     set address of data-block-ptr
360                                  to address of data-block
361     perform call-dialog-system
```

Moves the procedure `init-proc` to `ds-procedure` to show the first window. When control is returned, sets the address of the Data Block pointer.

Lines 365-369:

```
365     move ds-screenset-id to instance-screenset-id
366     move group-index to
367     group-record-no(ds-screenset-instance)
368   end-if
369      .
```

Stores the screenset-id and the line of the group this screenset instance is dealing with.

Lines 370-376:

```
370   close-instance section.
371
372     move ds-quit-set to ds-control
373     move 0 to group-record-no(ds-screenset-instance)
374   .
```

Closes a particular screenset instance.

Lines 377-389:

```
377   update-details section.
...
382      move group-record-no(ds-screenset-instance) to
383                                  group-index
384      move group-index to customer-index-of-interest
385      move redef-block
                     to customer-group-001-item(group-index)
...
389      move 0 to group-record-no(ds-screenset-instance)
```

Copies the information from the screenset instance to the main screenset and sets `customer-index-of-interest` to enable the screenset to update the correct group occurrence. It then clears the value of the instance table by moving 0 to the `group-record-no`.

Lines 394-401:

```
394   perform derivations
395     move ds-use-set to ds-control
396     move main-screenset-id to ds-set-name
397     move "refresh-proc" to ds-procedure
398
399     set address of data-block-ptr to
400       address of customer-data-block
401   .
```

Reinstates the main screenset and causes a refresh of the list box.

Lines 408-422:

```
404   close-all-instances section.
...
408     move 0 to group-index
409     move 1 to ignore-error
410     perform 10 times
411
412     add 1 to group-index
413     if group-record-no(group-index) not = 0
414       move instance-screenset-id to ds-set-name
415       move group-index to ds-screenset-instance
416       move ds-use-instance-set to ds-control
417       move "terminate-proc" to ds-procedure
418       perform call-dialog-system
419        move 0 to group-record-no(group-index)
420     end-if
421
422     end-perform
```

Closes all active screenset instances.

Lines 427-428:

```
427     set address of data-block-ptr to
428       address of customer-data-block
```

After closing all active screenset instances, reinstates the main screenset and sets the Data Block pointer.

18 Tutorial - Creating a Sample Screenset

In this chapter you follow the steps for creating an application using Dialog System as described in the chapter *Introduction to Dialog System*.

In this tutorial, imagine you are a director of a road running race that members of the public can enter by sending in entry forms published in various newspapers and sporting magazines. You need to design an interface for a system that will process the entries and allocate a unique running number to each entrant. You also want to monitor the effectiveness of your advertising by recording the response from each advertisement.

The first screen of your interface needs to collect details such as name, address, age, sex, running club, and advertisement code.

To create the screenset for the race entry system, first start Dialog System in the usual way.

18.1 The Sample Data Definition

You do not need to create the data model for the sample screenset. A data model has been created for you, and this section will guide you through defining the data according to that model.

18.1.1 Defining the Data Block

You need to define the Data Block and create master fields for the display fields you will set up as objects. Each entry field has an associated master field in which to store the data entered.

To define the Data Block:

1 Select **Data Block** on the **Screenset** menu.

Dialog System displays the Data Definition window.

2 Select **Prompted mode** on the **Options** menu.

The Data Type dialog box is shown.

3 Select **Field** and click **OK**.

Dialog System displays the Field Details dialog box.

4 Specify **NAME**, select **Alphanumeric**, and specify 15 in **Int**.

This defines the Name field and is the master field for the object called D-NAME that you will define in the next step.

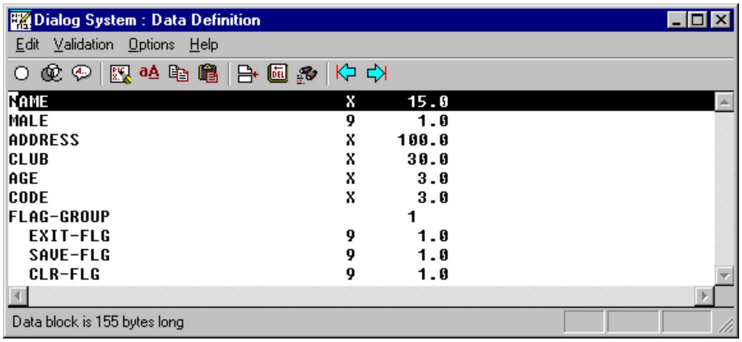
5 Click **OK** to create a blank line ready for the next entry.

Now repeat the above process to define the following fields:

Field Name	**Field Type and Size**	**Field Usage**
MALE	9(1)	flag field used to hold the state of the radio buttons.
ADDRESS	X(100)	master field for D-ADDRESS.
CLUB	X(30)	master field for D-CLUB.
AGE	X(3)	used in the age list box.
CODE	X(3)	master field for D-CODE.
FLAG-GROUP		Group start with 1 repeat. Used to hold flags.
EXIT-FLG	9(1)	flag used to indicate user has selected **Exit**.
SAVE-FLG	9(1)	flag used to indicate user has pressed **Save**.
CLR-FLG	9(1)	flag used to indicate user has pressed **Clear**.

This completes the Data Block for the sample program. The Data Definition window should now look similar to the one shown in Figure 18-1.

Figure 18-1. Data Definition



```
NAME                                 X     15.0
MALE                                 9      1.0
ADDRESS                              X    100.0
CLUB                                 X     30.0
AGE                                  X      3.0
CODE                                 X      3.0
FLAG-GROUP                                1
  EXIT-FLG                           9      1.0
  SAVE-FLG                           9      1.0
  CLR-FLG                            9      1.0
```

Data block is 155 bytes long

18.1.2 Creating the Sample Window Object

The sample screenset has a single window containing various control objects, and a message box. You must define the window first, because you do not have access to control objects until you have defined a window on which to place them.

Before you start to define objects, set **Auto properties** to off, so that you can label objects as you create them:

1. Select **Include** on the **Options** menu on the main menu bar.
2. Select **Auto properties** on the drop down menu.

To define the Primary window:

1. Select the primary window icon on the **Objects** toolbar or choose **Primary window** on the **Object** menu.
2. Move the window box until the top left-hand corner is near the top left of your desktop.
3. Click to fix the position.

4 Enlarge the window by moving the mouse down and right.

Make the window larger than the Dialog System window.

5 Click to fix its size.

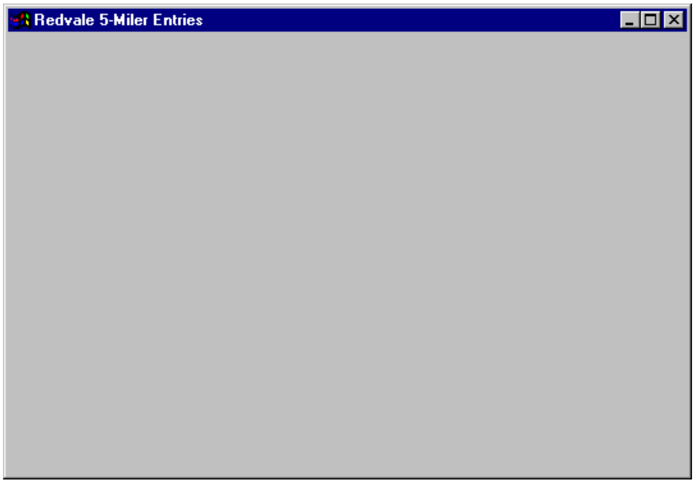
6 When the Properties dialog box appears, specify a name of **MAIN** and a title of **Redvale 5-Miler Entries**.

7 Select the **Options** tab and deselect **Menu**.

8 Leave the rest of the dialog box unaltered to accept the default property values.

The screen now looks like the one shown in Figure 18-2.

Figure 18-2. Defining the Main Window



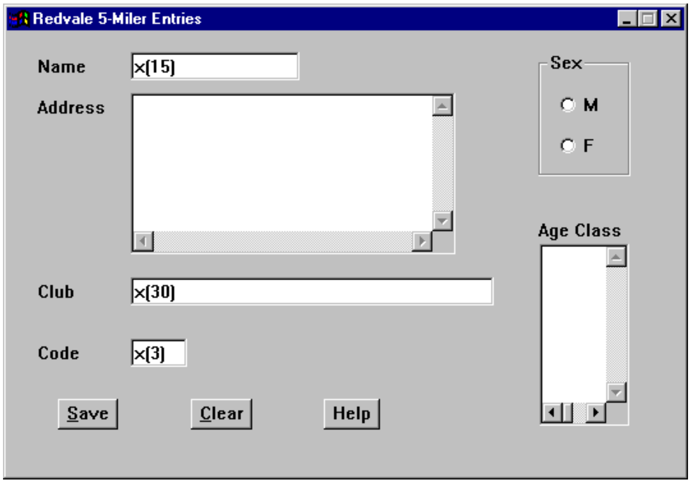
18.1.3 Creating the Sample Control Objects

Now define the controls in the following list by:

1 Selecting them on the **Objects** toolbar or the **Object** menu.

2 Placing them in the primary window, to look like Figure 18-3.

Figure 18-3. Adding Controls



3 Making the indicated changes to their properties in their property dialog boxes.

Leave all the other properties as the default values.

Text — Label for the competitor name field. Specify **Name**.

Entry field — To contain the competitor's name. Specify **D-NAME** in **Name**, **X(15)** in **Picture** and **NAME** in **Master**.

Text — Label for the competitor address field. Specify **Address**.

Multiple line entry field — To contain the competitor's address. Specify **D-ADDRESS** in **Name**, **120** in **Length** and **ADDRESS** in **Master**.

Text — Label for the competitor's club field. Specify **Club**.

Entry field — To contain the competitor's running club. Specify **D-CLUB** in **Name**, **X(30)** in **Picture** and **CLUB** in **Master**.

Radio button — To show the competitor's sex. Specify **M** in **Text** and set the Initial state to **Enabled**.

Radio button	To show competitor’s sex. Specify **F** in **Text** and set the Initial state to **Enabled**.
Group box	Position over the two radio buttons. Specify **Sex** in **Text**.
Text	Label for age class field. Specify **Age class**.
List box	To show the competitor’s age class. Select **Initial Text Defined**. Click on **List text** and specify the age classes, one per line, as follows: **10-18**, **19-35**, **36-40**, **41-45**, **46-50**, **51-55**, **56-60**, **60+**. Deselect **Horizontal scroll bar** check box.
Text	Label for advertisement code field. Specify **Code**.
Entry field	To contain an advertisement code. Specify **D-CODE** in **Name**, **X(3)** in **Picture** and **CODE** in **Master**.
Push button	Button to enter the data into the competitor data base. Specify **"Save"** in **Text**, enable **Default Button** and set the Initial State to **Enabled**.
Push button	Button to clear the current entry. Specify **Clear** in **Text** and set the Initial State to **Enabled**.
Push button	Button to get help. Specify **Help** in **Text** and set the Initial State to **Enabled**.

Move and size the controls (and the primary window if necessary) to tidy your window.

To make the radio buttons work correctly, you need to place them in a control group. Similarly, the push buttons should be in a control group.

To define control groups:

1 Select **Edit**.

2 Select **Define control group**.

3 Position the top left corner of the box that appears.

4 Click and extend the box until it covers the required controls.

5 Click again.

18.1.4 Creating a Message Box

Create a message box to present a help message when the user presses **Help**:

1 Select a message box object on the **Objects** toolbar or the **Object** menu.

The Properties dialog box is displayed immediately.

2 Specify **HELP-MSG** in **Name**.

3 Specify **Help** in **Heading**.

4 Specify some suitable help text about the window in **Text**.

5 Select **Information** on the **Icon** drop down list.

This completes the object definitions for the sample screenset.

18.1.5 Saving Your Screenset

It is a good idea to save a screenset whenever you have finished a stage in the definition process. To save this sample screenset:

1 Select **Save As** because this is the first time you have saved it.

Dialog System displays a dialog box for you to enter the filename and directory under which you want to save.

2 Specify **entries.gs** as the name.

After saving the sample screenset once, you can use **Save** whenever you want to save the screenset. If you want to try out different versions of the screenset, use **Save As** with a new name to create another version.

18.1.6 Testing

You can test your screenset even without dialog, to make sure that the desktop layout looks good and that you have set up the fields correctly. To test the sample screenset:

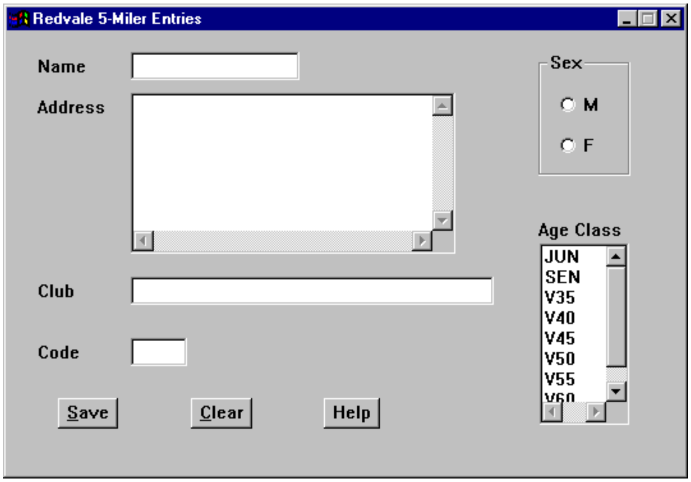
1 Select **Debug** on the **File** menu.

Dialog System displays the Screenset Animator window.

2 Now select **Run** on the **Execute** menu.

Your sample screenset should look similar to the one shown in Figure 18-4.

Figure 18-4. Running the Screenset from the Screenset Animator



Try entering some data into the fields. If you try to enter more than 15 characters into the Name field, Dialog System just produces a beep.

Notice that you can only select either M or F because the radio buttons are grouped as a control group and you can select only one age class from the list box.

3 Press **Esc** to return to the Screenset Animator window.

The "RETC has just been executed" dialog box is shown.

4 Click **Interrupt**.

5 Select **Exit** on the **Execute** menu to return to Dialog System's main window.

The Screenset Animator is more useful after you have defined the dialog for the sample screenset.

18.1.7 Defining Dialog

The sample screenset would not be complete without dialog definitions. The following sections explain how to define the object and global dialog for the sample screenset.

18.1.7.1 The Sample Object Dialog Definitions

To define dialog for an object:

1 Select the **Save** push button object.

2 Select **Object dialog** on the **Edit** menu.

Dialog System displays the Dialog Definition window. The default event **BUTTON-SELECTED** is already displayed in the Dialog Definition window.

3 Select **Prompted mode** on the **Options** menu.

Dialog System then prompts you for dialog entries in a similar way to when you define data.

4 Use the cursor key to move down one line.

Dialog System displays a dialog box for you to choose the type of line you want to enter.

5 Select **Comment** and enter some text explaining what your dialog does.

For example, "Dialog System passes control to a procedure that handles creating or changing a record in the entry database".

6 Specify **Action when SAVE pressed**.

7 Select **Function**.

8 Select **SET-FLAG** from the scrollable list.

A parameter dialog box appears.

9 Specify the name of the flag to be set against **parameter 1**.

The spaces for parameters 2 and 3 are disabled because they are not required with this function.

10 Specify **SAVE-FLG(1)** against **parameter 1**, to set the **Save** flag when the user presses **Save**.

11 Select **Function**.

12 Select **RETC** from the scrollable list.

RETC does not have any parameters. It returns control to the calling program. The COBOL program that uses this screenset checks the **Save** flag and saves the contents of the screen if the flag is set.

In a similar way you can attach dialog to the other objects in the sample screenset.

Dialog System helps you by providing context menus. If you right-click on any of the dialog lines, Dialog System displays a context-sensitive menu of choices. For a more detailed explanation of context menus, see the chapter *Window Objects*.

The dialog for the remaining objects is described below.

18.1.7.1.1 Clear Button

Dialog:

```
BUTTON-SELECTED
    SET-FLAG CLR-FLG(1)
    RETC
```

Effect: When the user presses **Clear,** Dialog System sets the Clear flag and returns control to the calling program. The calling program will clear the entry fields (effectively cancelling the entry).

18.1.7.1.2 Help Button

Dialog:

```
BUTTON-SELECTED
    INVOKE-MESSAGE-BOX HELP-MSG $NULL $EVENT-DATA
```

Effect: When the user presses **Help**, Dialog System displays the message box you created. This provides appropriate help text.

18.1.7.1.3 M Radio Button

Dialog:

```
BUTTON-SELECTED
    SET-FLAG MALE
```

Effect: When the user presses **M**, Dialog System sets the flag **MALE** to **1**, meaning M (for male competitor).

18.1.7.1.4 F Radio Button

Dialog:

```
BUTTON-SELECTED
    CLEAR-FLAG MALE
```

Effect: When the user presses **F**, Dialog System sets the flag **MALE** to **0**, meaning F (for female competitor).

18.1.7.1.5 Age Class List Box

Dialog

```
ITEM-SELECTED
    RETRIEVE-LIST-ITEM $CONTROL AGE $EVENT-DATA
```

Effect When the user selects an age class from the list, Dialog System puts the value of the selected list item into the **AGE** field.

18.1.7.2 The Sample Global Dialog Definition

Now define global dialog for the screenset, by selecting **Global dialog** on the **Screenset** menu.

The default global dialog is as follows:

```
ESC
    RETC
CLOSED-WINDOW
    RETC
```

Add two lines to the dialog so it reads:

```
ESC
    SET-FLAG EXIT-FLG(1)
    RETC
CLOSED-WINDOW
    SET-FLAG EXIT-FLG(1)
    RETC
```

This dialog sets the exit flag and returns control to your application (the calling) program if the user presses **Esc** or uses the system menu to close the window.

Add the following dialog:

```
  REFRESH-DATA
    REFRESH-OBJECT MAIN
```

This dialog refreshes the main window whenever the calling program directs Dialog System to do so. The chapter *Using the Screenset* describes how your application program interfaces with the call interface.

Add the following dialog:

```
  SCREENSET-INITIALIZED
    SET-FLAG MALE
    SET-BUTTON-STATE RB1 1
```

This sets the first radio button (Male) on as a default. Whenever you enter a group of radio buttons, one of these has to be set on initially.

See the chapter *Using Dialog* for more information on how to use dialog.

18.1.8 Testing the Screenset Again

Now save the screenset again, then try running the sample again. Enter data into the fields and choose the appropriate radio button and list item. This time, when you press **Save** after you have made these changes, Dialog System displays the Screenset Animator window again.

18.1.9 Changing the Screenset

After you test the sample screenset again, you may want to make changes to it (for example, to improve the screen layout). You can repeat any of the steps described in this chapter until you are satisfied with the screenset.

18.1.10 Summary

In this chapter, you have:

- Created a sample screenset.
- Defined data.
- Defined objects.
- Saved the screenset.
- Tested the screenset.
- Added dialog to the screenset.
- Tested it again.
- Made any changes by repeating steps as necessary.

Now you need to write the COBOL program that will use the user interface contained in your sample screenset. The **DialogSystem\demo\entries** directory contains a demonstration version of the sample screenset, called **entriesx.gs**.

18.2 Further Information

The next chapter, *Tutorial - Using the Sample Screenset*, gives advice on writing the COBOL program for the sample screenset. The chapter contains sample code to produce a very simple program that will read the user inputs and store or clear them as indicated by the user.

19 Tutorial - Using the Sample Screenset

The chapter *Tutorial - Creating a Sample Screenset* explained how to create the Entries screenset containing the user interface for a sample application. This chapter explains how to write an application program to use the Entries screenset, and guides you through the following steps:

1. Generating the COBOL copyfile from the screenset.
2. Writing the COBOL application program with the necessary calls to the Dialog System run-time.
3. Debugging and animating the COBOL program.
4. Packaging your application.

19.1 Generating the Data Block Copyfile

The Data Block copyfile contains the definition of the Data Block passed from the calling program to Dialog System at run time. You must include the copyfile in your calling program. The copyfile also contains version checking information.

Dialog System enables you to generate a copyfile from your screenset, and set options determining how that copyfile is generated.

To create the copyfile for your sample screenset, first set the screenset configuration options, then generate the copyfile:

1. Select **Configuration** on the **Options** menu.
2. Select **Screenset** on the popup menu.
3. Enter the name **Entry** in **Screenset identifier**.
4. Check that **Fields prefixed by screenset ID** check box is on.

This specifies that all the data names from the data block are prefixed by the string "entry". The sample program uses these prefixed data names.

5 Select **Enter** or **OK** to accept the other defaults from this dialog box.

6 Select **Generate** on the **File** menu.

7 Select **Data block COPY-File** from the drop down menu that appears.

8 Enter the name of the file to be used for the copyfile - **entries.cpb**.

9 Select **Enter**.

Dialog System generates the copyfile for the sample screenset using the copyfile options you have just set.

19.1.1 Selecting Options and Generating the Copyfile

You have already generated the sample copyfile.

19.2 Writing the COBOL Application Program

A COBOL program that uses the sample screenset you created in the previous chapter is shown below. This program is provided with your Dialog System software as a demonstration program. It is called **entries.cbl**.

Your requirements and programming style may result in a different program structure.

```
$set ans85
 identification division.

 program-id. race-entries.

 environment division.

 input-output section.
 file-control.
```

```
     select entry-file assign "entries.dat"
         access is sequential.

 data division.

 file section.
 fd  entry-file.
 01  entry-record.
     03  file-name              pic x(15).
     03  file-male              pic 9.
     03  file-address           pic x(100).
     03  file-club              pic x(30).
     03  file-code              pic x(3).

 working-storage section.
     copy "ds-cntrl.v1".
     copy "entries.cpb".

 78  refresh-text-and-data-proc value 255
 77  display-error-no           pic 9(4).

 procedure division.

 main-process section.
     perform program-initialize
     perform program-body until entry-exit-flg-true
     perform program-terminate.

 program-initialize section.
     initialize entry-data-block
     initialize ds-control-block
     move entry-data-block-version-no
                           to ds-data-block-version-no
     move entry-version-no to ds-version-no
     open output entry-file
    perform load-screenset.

 program-body section.
* Process the returned user action (in the flags); clear
* those flags and call Dialog System again.

     evaluate true
       when entry-save-flg-true
         perform save-record
       when entry-clr-flg-true
         perform clear-record
     end-evaluate
     perform clear-flags
     perform call-dialog-system.

 program-terminate section.
     close entry-file
     stop run.

 save-record section.
* Save the current details in the file if all the text
* fields contain values.

     if (entry-name    < > spaces) and
        (entry-address < > spaces) and
        (entry-club    < > spaces) and
        (entry-code    < > spaces)
            move entry-name    to file-name
            move entry-male    to file-male
            move entry-address to file-address
            move entry-club    to file-club
            move entry-code    to file-code
            write entry-record
     end-if.

 clear-record section.
* Clear the current details by initializng the data block

     initialize entry-record
     initialize entry-data-block
     perform set-up-for-refresh-screen.

 clear-flags section.
     initialize entry-flag-group.

 set-up-for-refresh-screen section.
* Force Dialog System to execute the procedure P225 (in
* global dialog) the next time it is called. This
* procedure simply refreshes the main window with the
* values from the data block.

     move "refresh-data" to ds-procedure.

 load-screenset section.
* Specify the screenset to be used and call Dialog System

     move ds-new-set to ds-control
     move "entries"  to ds-set-name
     perform call-dialog-system.
```

```
82  call-dialog-system section.
83      call "dsgrun" using ds-control-block,
84                          entry-data-block.
85      if not ds-no-error
86          move ds-error-code to display-error-no
87          display "ds error no:   " display-error-no
88          perform program-terminate
89      end-if.
```

Lines 1 to 22:

```
1  $set ans85
2  identification division.

3  program-id. race-entries.

4  environment division.

5  input-output section.
6  file-control.
7      select entry-file assign "entries.dat"
8      access is sequential.

9  data division.

10 file section.
11 fd  entry-file.
12 01  entry-record.
13     03  file-name              pic x(15).
14     03  file-male              pic 9.
15     03  file-address           pic x(100).
16     03  file-club              pic x(30).
17     03  file-code              pic x(3).

18 working-storage section.
19     copy "ds-cntrl.v1".
20     copy "entries.cpb".

21 78  refresh-text-and-data-proc value 255
22 77  display-error-no              pic 9(4).
```

These lines set up the records for storage of the entries input by the user.

Lines 23 to 27:

```
23  procedure division.

24  main-process section.
25      perform program-initialize
26      perform program-body until entry-exit-flg-true
27      perform program-terminate.
```

These lines structure the whole program, making sure that it ends when the user presses **Esc** or uses the System menu to close the window. These actions set the Exit flag in the screenset. The flag setting is passed to the program for action.

Lines 28 to 35:

```
28  program-initialize section.
29      initialize entry-data-block
30      initialize ds-control-block
31      move entry-data-block-version-no
32                            to ds-data-block-version-no
33      move entry-version-no to ds-version-no
34      open output entry-file
35      perform load-screenset.
```

The Data Block copyfile contains not only the user data but some version numbers that Dialog System checks against the screenset when it is called. To do this, the calling program must copy them into data items in the Control Block before it calls the Dialog System run-time.

When you write your calling program, you must copy the copyfile into the program Working-Storage Section using the statement: *copy "ds-cntrl.mf"*. If you are using ANSI-85 conformant COBOL then you should use the copyfile ds-cntrl.ans.

You also need to make sure that the Control Block contains the name of the screenset and other information that controls Dialog System behavior.

Lines 36 to 46:

```
36  program-body section.
37 * Process the returned user action (in the flags); clear
38 * those flags and call Dialog System again.

39      evaluate true
40        when entry-save-flg-true
41          perform save-record
42        when entry-clr-flg-true
43          perform clear-record
44      end-evaluate
```

```
45      perform clear-flags
46      perform call-dialog-system.
```

Dialog System enables the user to decide which functions the program is to perform rather than the program dictating the user's actions. The flags set in the Data Block returned from Dialog System contain values resulting from the user's action. These values tell the program what to do next.

The program can respond in a variety of ways including:

- Modifying stored information.

  Done in the `save-record` section.

- Retrieving additional information from a database.

  Not done in this application.

- Displaying results or error messages.

  Done in the `call-dialog-system` section, if an error occurs when the program calls Dialog System.

- Providing assistance in the use of the application.

  In this simple application, help is handled entirely by a message box in Dialog System. An alternative way of handling help would be to write a section of code in the program to provide help whenever a Help flag was set.

- Validating the information entered by the user.

  A minimal level of validation is done in the `save-record` section to ensure that empty records are not saved. Obviously, more complex validation can be done both by the program and by Dialog System itself at the input stage.

- Requesting additional input from the user.

  By returning to Dialog System after saving or clearing the record.

Lines 47 to 49:

```
47  program-terminate section.
48      close entry-file
49      stop run.
```

These lines end the program when an error occurs or when the user presses **Esc** or uses the System menu to close the window.

Lines 50 to 63:

```
50  save-record section.
51 * Save the current details in the file if all the text
52 * fields contain values.

53      if (entry-name    < > spaces) and
54         (entry-address < > spaces) and
55         (entry-club    < > spaces) and
56         (entry-code    < > spaces)
57             move entry-name    to file-name
58             move entry-male    to file-male
59             move entry-address to file-address
60             move entry-club    to file-club
61             move entry-code    to file-code
62             write entry-record
63      end-if.
```

These lines save the user's input if the user has pressed **Save**. Pressing this button sets the Save flag which the program tests in the `program-body` section. The input is not saved if the record is empty.

Lines 64 to 68:

```
64  clear-record section.
65 * Clear the current details by initializing the data block.

66      initialize entry-record
67      initialize entry-data-block
68      perform set-up-for-refresh-screen.
```

These lines clear the current inputs from the screen and from the Data Block when the user presses **Clear**. Pressing this button sets the Clear flag which the program tests in the `program-body` section.

Lines 69 to 76:

```
69  clear-flags section.
70      initialize entry-flag-group.

71  set-up-for-refresh-screen section.

72 * Force Dialog System to execute the procedure P225 (in
73 * global dialog) the next time it is called. This
74 * procedure simply refreshes the main window with the
75 * values from the data block.

76      move "refresh-data" to ds-procedure.
```

These lines clear the flags and tell Dialog System to refresh the screen ready for the next user inputs.

Lines 77 to 89:

```
77  load-screenset section.
78 * Specify the screenset to be used and call Dialog System

79      move ds-new-set to ds-control
80      move "entries"  to ds-set-name
81      perform call-dialog-system.

82  call-dialog-system section.
83      call "dsgrun" using ds-control-block,
84                          entry-data-block.
85      if not ds-no-error
86          move ds-error-code to display-error-no
87          display "ds error no:   " display-error-no
88          perform program-terminate
89      end-if.
```

These lines load the correct screenset and call Dialog System. The second section also checks for calling errors and ends the program if an error occurs.

19.3 Debugging and Animating the COBOL Program

Your COBOL system provides an editing, debugging and animating environment.

When you are debugging an application, the source code of each program is displayed in a separate window. When you animate the code, each line of the source is highlighted in turn as each statement is executed, showing the effect of each statement. You can control the pace at which the program executes and can interrupt execution to examine and change data items. See the topic *Debugging* in the Help for more information.

19.4 Packaging Your Application

To create the finished application you must complete various subtasks.

You use the Project facility from within Net Express to build your Dialog System application. See the topic *Building Applications* in your Help for further details.

The topic *Compiling* in your Help explains what you must do next to prepare your application for production.

When testing is completed, you are ready to assemble the finished product. A finished product can be copied onto diskettes, sent to a customer, and loaded onto another machine to run as an application.

Depending on the size of the application, the finished product consists of one or more of the following:

- Executable modules. These are in the industry standard **.exe** and **dll** file format.
- Run-time support files. These files perform functions such as file I/O and memory management.

20 Tutorial - Adding and Customizing a Status Bar

This tutorial takes you through adding the status bar control program to the sample Customer screenset supplied with Dialog System. You use the same steps to add a status bar to any window in any screenset. The same principles apply to adding any other control program to a screenset. You will find out how to:

- *Add a status bar to the screenset.*
- *Run the screenset.*
- *Manipulate the status bar.*
- *Register Events for the Status Bar*

Before starting this tutorial you should have:

- Read through the chapter *Programming Your Own Controls*.

The user control objects and control programs are also documented in:

- *Objects and Properties* in the Help.

  The topic on user controls describes how to create a user control and set its properties.

- *Control programs* in the Help.

  The topics in this section give detailed information on the control programs and the functions available.

- The COBOL source code itself.

A separate demonstration of using a status bar control program is provided in the screenset **sbards.gs**, and is documented in the file **sbards.txt** in your **DialogSystem\demo\sbards** subdirectory.

20.1 Setting Up

1 You will be using the Customer screenset, and making many changes to it, so we advise you to make a backup of **customer.gs** before starting.

2 Start Net Express.

3 Open **customer.app** in your **DialogSystem\demo\customer** subdirectory.

4 Right-click on **customer.gs** in the left-hand pane, and select **Edit** from the context menu.

Dialog System starts up and the Customer Details window is displayed.

20.2 Adding a Status Bar to the Screenset

Before using any of the control programs, a common data area must be defined in the screenset Data Block. This is used to pass information between a screenset and the control programs at run time.

20.2.1 Defining the Data Items

Each status bar that you define in a screenset must have a Data Block item (master field) associated with it. The data item must be defined as `OBJ-REF` as it is used to hold the class library object reference of the created control. The data item must be defined before you add the status bar. If you do not have any object references defined in your screenset, then Dialog System will not let you define a status bar.

Include the following data item in the Data Block of the screenset, to hold the object reference of the status bar:

```
MAIN-WINDOW-SBAR-OBJREF          OBJ-REF
```

Include the following data definition for FUNCTION-DATA in the screenset Data Block:

```
FUNCTION-DATA                              1
  WINDOW-HANDLE                     C5     4.0
  OBJECT-REFERENCE                  OBJ-REF
  CALL-FUNCTION                     X     30.0
  NUMERIC-VALUE                     C5     4.0
  NUMERIC-VALUE2                    C5     4.0
  SIZE-WIDTH                        C5     4.0
  SIZE-HEIGHT                       C5     4.0
  POSITION-X                        C5     4.0
  POSITION-Y                        C5     4.0
  IO-TEXT-BUFFER                    X    256.0
  IO-TEXT-BUFFER2                   X    256.0
```

The FUNCTION-DATA definition can also be imported from the file **funcdata.imp** in the **DialogSystem\source** subdirectory. Select **File**, **Import**, **Screenset** and click **OK** to acknowlege the currently loaded screenset might be overwritten. Click the **File** button, double-click **funcdata.imp**, click the **Import** button and then **OK** and **Close**.

Since FUNCTION-DATA is common to all the control programs, you need define it only once in each screenset that uses the control programs.

20.2.2 Defining the Status Bar

When you have defined the required data, define the status bar:

1. Select the Customer Details window, MAIN-WINDOW, to which you want to add the status bar.
2. Select **Status Bar** on the **Programmed Controls** toolbar.
3. Position and size the status bar.

   The status bar is positioned at the bottom of the window, regardless of where it was positioned when it was painted.
4. Complete the following items on the Status Bar Properties dialog box:

   a. Choose an appropriate name for the status bar. Make this as descriptive as possible, for example MAIN-WINDOW-STATUS-BAR.

b Specify **MAIN-WINDOW-SBAR-OBJREF** as the master field name.

c Specify **SBAR2** as the name of the status bar control program.

d Check Add program to current project.

e Click **Generate** to generate the status bar control program **sbar2.cbl**.

A message box appears, reminding you that you need to include the necessary entries in the Data Block for use by the generated control program. You have done this in the earlier steps, so click **OK** to continue.

You can see the generated program being added to your Customer project in the background.

5 Click **OK**, save the screenset, and close it.

6 Return to Net Express, right-click on **customer.gs**, and select **Generate copyfile** from the context menu.

7 Compile the generated control program using **Rebuild All** on the Net Express **Project** menu to obtain an executable version of this program.

20.3 Running Your Screenset

Once you have completed all these steps with your screenset, you can run it to see the User Control working.

At this stage, the status bar will not do anything useful - the clock and key states on the status bar will not update.

You need to add dialog for the status bar updating to work correctly.

20.4 Manipulating the Status Bar

Each user control can be manipulated using the predefined functions in its control program. By setting FUNCTION-NAME and other parameters in FUNCTION-DATA you can perform actions on the control such as refreshing, deleting or updating data associated with it.

The following sections take you through implementing code that will maintain the state of information displayed in the status bar, including:

- Clock time.
- Insert/Caps/NumLock Key state information.
- Window/Status Bar section resizing.
- Mouse-over hint text.

20.4.1 Clock Time and Key State Maintenance

In order to maintain the accuracy of the clock and key state information, the status bar needs to be refreshed regularly. To do this, you need to use the timeout facility of Dialog System.

20.4.1.1 Using the Timeout Facility

To set the timeout, you need to add the WINDOW-CREATED event to the window containing the status bar (in this case MAIN-WINDOW).

1. Select the Customer Details window and go to **Object Dialog**.
2. Click the **Event** toolbar button and select WINDOW-CREATED.
3. Add the following line of dialog:

   ```
   TIMEOUT 25 REFRESH-STATUS-BAR
   ```

   This dialog causes the REFRESH-STATUS-BAR procedure (which will be defined later) to be executed once every quarter of a second.

Note: If you try to debug the application using Screenset Animator, Screenset Animator will appear to execute the REFRESH-STATUS-BAR procedure in a loop. This is because it is difficult to interact with the application in the time between the last line of the REFRESH-STATUS-BAR being executed, and the time when the next timeout event is triggered (one quarter of a second later). For this reason, you may wish to adjust the timeout value, or remove this line altogether when debugging with Screenset Animator.

4 Next, you need to add the dialog to process the timeout event. In general, the procedure which is triggered when a timeout event occurs should be placed in global dialog. If you place the dialog on an object, and that object loses focus, the Dialog System run-time might no longer be able to find the timeout procedure (which will cause run-time error 8 to be generated).

The following dialog causes the status bar to be updated when the timeout event occurs (using the REFRESH-OBJECT function of the status bar control program). You must place this dialog in global dialog for it to work correctly.

```
REFRESH-STATUS-BAR
  MOVE "REFRESH-OBJECT" CALL-FUNCTION(1)
  SET OBJECT-REFERENCE(1) MAIN-WINDOW-SBAR-OBJREF
  CALLOUT "SBAR2" 0 $NULL
```

20.4.2 Window/Status Bar Section Resizing

Next, you need to add dialog to resize the status bar correctly when the window is resized. The status bar control program has a RESIZE function which you need to call when the window is resized, maximized, or restored.

You do not need to call the function when the window is minimized, because the status bar is not visible when the window is minimized.

Note: In order to resize the window, it must have the Size Border property set.

Go to **Object Dialog** on the toolbar and add the following dialog to the window containing the status bar, to cause the status bar to resize when the window resizes:

```
WINDOW-SIZED
    BRANCH-TO-PROCEDURE RESIZE-PROCEDURE

WINDOW-RESTORED
    BRANCH-TO-PROCEDURE RESIZE-PROCEDURE

WINDOW-MAXIMIZED
    BRANCH-TO-PROCEDURE RESIZE-PROCEDURE

  RESIZE-PROCEDURE
    MOVE "RESIZE" CALL-FUNCTION(1)
    SET OBJECT-REFERENCE(1) MAIN-WINDOW-SBAR-OBJREF
    CALLOUT "SBAR2" 0 $NULL
```

20.4.3 Adding Mouse-over Hint Text

To have a fully functioning status bar, you need to update the text that appears in the status bar whenever the mouse moves over objects on the window. To do this you need to:

1 Enable MOUSE-OVER events for the window containing the status bar (using the SET-PROPERTY dialog function).

Add the following line of dialog to the WINDOW-CREATED event:

```
SET-PROPERTY MAIN-WINDOW "MOUSE-OVER" 1
```

You will also need to add this dialog to any other window on which you require MOUSE-OVER events.

2 Respond to the MOUSE-OVER events (when they are generated) by adding dialog to set the text on the status bar. The MOUSE-OVER text usually goes in the leftmost section (section number one) on the status bar.

Add the following dialog procedure to the global dialog:

```
DISPLAY-HINT-TEXT
  MOVE "UPDATE-SECTION-TEXT" CALL-FUNCTION(1)
  MOVE 1 NUMERIC-VALUE(1)
  SET OBJECT-REFERENCE(1) MAIN-WINDOW-SBAR-OBJREF
  CALLOUT "SBAR2" 0 $NULL
```

3. Update the status line whenever a MOUSE-OVER event occurs on an object by adding dialog similar to the following for each object on the window which needs to update the status text:

   ```
   MOUSE-OVER
     MOVE "Status text" IO-TEXT-BUFFER(1)
     BRANCH-TO-PROCEDURE DISPLAY-HINT-TEXT
   ```

   Replace the words *Status text* with the text you want to appear on the status bar.

   For example, replace *Status Text* with *Load Record* for the LOAD push button on MAIN-WINDOW of the CUSTOMER screenset.

If you now run the screenset, you will see that the CUSTOMER screenset has a status bar which:

- Displays MOUSE-OVER text in section 1 of the status bar.
- Displays the key states of the INSERT, CAPS LOCK and NUM LOCK keys in sections 2, 3 and 4 of the status bar.
- Displays the current time in section 5 of the status bar.

Click **Abort** when you have finished looking at the screenset, save it and close Dialog System.

The next section shows you how to register events for the status bar.

20.5 Registering Events for the Status Bar

The status bar control program has code to handle the event generated when the user clicks the mouse over one of the sections on the status bar. The section *Customizing the Status Bar Control Program* shows how to customize the status bar control program to add another event.

The class library sends events to your application by means of a *callback*. For example, the status bar control program registers its entry point **SBar2Button1Down** with the class library so that whenever the user clicks the left mouse button over the status bar, the code in the entry point **Sbar2Button1Down** is executed.

Callback registration is performed after full creation of the control in the Create Entrypoint of the generated program. You should register a callback in the "Register-callbacks" section for all events for which you need to provide code.

When an event is generated, the callback code associated with the event can update the data in your Data Block to pass back to the Dialog System screenset. See the topic *Control programs* in the Help for information on how to pass data between the control programs and the Dialog System screenset.

This completes the steps required to add a status bar control program to any Dialog System screenset.

You can use similar steps to these to add any control program to any screenset, tailoring the dialog to your requirements.

20.6 20.6 Customizing the Status Bar Control Program

This part of the tutorial describes how to add functionality to a status bar so that when a user double-clicks on the clock section, you can receive a callback to process the event as required.

The status bar control program which you generated at definition time may be used ’as is’ and is coded in such a way as to allow the creation and maintenance of many controls on as many windows as you require. You can however, add your own features.

Note: Each screenset that contains a status bar must use its own generated status bar control program. This is because each generated status bar control program contains code specific to the screenset for which it was generated.

For example, you might want to add an extra section to the status bar. As an example, this section shows you how to tailor the status bar control program to add an extra event.

For the purpose of this example, it is assumed you have generated a control program called **sbar2.cbl**, which you will now customize.

In all cases, you will need to ensure your tailored program is available on $COBDIR at run time.

20.6.1 Registering a Callback for the New Event

To add the left-mouse-button-double-click event to the status bar, you need to do two things:

- Add code to the Register-Callbacks section to register the new event.
- Add code to the control program to process the new event.

Registering a callback for the new event is very similar to the code for registering the callback for the left-mouse-button-down event. Before adding the callback registration code for the new event, it is worth explaining how the code for the new event works:

```
MOVE ProgramID & "Button1Down " TO MessageName
```

Note: The Program ID constant is used in the naming of the entry points in this program. This prevents different control programs loading duplicate names.

Stores the name of the entry point that is to receive the callback (in this case, Button1Down) in MessageName.

```
INVOKE EntryCallback "new" USING MessageName
                           RETURNING aCallback
```

Creates a new callback object, using the entry point name specified above.

```
MOVE p2ce-Button1Down TO i
```

Stores the event number (p2ce-Button1Down) in the variable I. All of the available event numbers are listed in the file **source\guicl\p2cevent.cpy** in your Net Express system.

```
INVOKE aStatusBar "setEvent" USING i aCallback
```

Sets the event (p2ce-Button1Down) to cause a callback to the entry point (*ProgramID* & "Button1Down").

```
INVOKE aCallback "finalize" RETURNING aCallback
```

The callback object is no longer required, so it can be deleted (finalized).

You can now add the code to register the callback for the new event. This is done by adding code to the Register-Callbacks section of the control program.

To add your left-mouse-button-double-click registration code:

1 Click on Net Express.

2 Right-click on **sbar2.cbl** and select **Edit** from the context menu.

3 Insert the code below immediately after the registration code for the left-mouse-button-down event, before the comments.

```
MOVE ProgramID & "DblClk1 " TO MessageName
INVOKE EntryCallback "new" USING MessageName
                       RETURNING aCallback
MOVE p2ce-Button1DblClk TO i
INVOKE aStatusBar "setEvent" USING i aCallback
INVOKE aCallback "finalize" RETURNING aCallback
```

20.6.2 Adding a Left-mouse-button-double-click Event

Since the left-mouse-button-double-click event is very similar to the left-mouse-button-down event, you can copy the existing code in the left-mouse-button-down section to add a new section which will process the event. The only code which is different between the events is the section and entry point names, and the user event number.

20.6.2.1 Adding the Code

To add the code to process the new event, open **sbar2.cbl** for editing as above, and:

1 Duplicate all of the code in the section left-mouse-button-down. Call the new section LEFT-MOUSE-BUTTON-DBLCLK.

2 Change the name of the entry point in the LEFT-MOUSE-BUTTON-DBLCLK section from Button1Down to DblClk1.

3 Change the user event number in the LEFT-MOUSE-BUTTON-DBLCLK section from 34590 to 34591.

4 Save **sbar2.cbl**.

5 Rebuild the project.

Note: Please keep a note of the user events used and generated by the control programs used in your applications, so that duplication does not occur. Although you are not prevented from using duplicate user events, if you use the same user event for two different purposes on the same window, your program might not work as expected.

20.6.2.2 Adding Additional Dialog

Once you have done this, you can add some additional dialog to make use of the new event that you have added to your customized status bar control program.

1 As an example, right-click **customer.gs** in Net Express, and select **Edit** from the context menu.

2 Right-click in the Customer Details window and select **Dialog** from the context menu.

3 To add the new event and make your machine beep when you double-click in the status bar, enter the following code:

```
USER-EVENT
   XIF= $EVENT-DATA 34591 SBAR2-DOUBLE-CLICK
 SBAR2-DOUBLE-CLICK
   BEEP
```

4 Save the screenset and run it.

5 Double-click in the status bar to test the beep.

The next chapter explains how to add and customize a menu bar and toolbar.

21 Tutorial - Adding and Customizing a Menu Bar and Toolbar

This tutorial takes you through adding the menu bar and toolbar control program to the sample Customer screenset supplied with Dialog System. You will notice that many of the steps for adding the menu bar are almost exactly the same as those for adding a status bar, as described in the previous tutorial.

You use the same steps to add a menu bar and toolbar to any window in any screenset. The same principles apply to adding any other control program to a screenset. You will find out how to:

- *Add a menu bar and toolbar to the screenset.*
- *Run the screenset.*
- *Define a menu structure.*
- *Define a toolbar structure.*
- *Customize the toolbar.*

Before starting this tutorial you should have:

- Read through the chapter *Programming Your Own Controls*.

The user control objects and control programs are also documented in:

- *Objects and Properties* in the Help.

  The topic on user controls describes how to create a user control and set its properties.

- *Control programs* in the Help.

  The topics in this section give detailed information on the control programs and the functions available.

- The COBOL source code itself.

A separate demonstration of using a toolbar control program is provided in the screenset **tbards.gs**, and is documented in the file **tbards.txt** in your **DialogSystem\demo\tbards** subdirectory.

21.1 Setting Up

1. You will be using the Customer screenset, and making many changes to it, so we advise you to make a backup of **customer.gs** before starting.
2. Start Net Express.
3. Open **customer.app** in your **DialogSystem\demo\customer** subdirectory.
4. Right-click on **customer.gs** in the left-hand pane, and select **Edit** from the context menu.

   Dialog System starts up and the Customer Details window is displayed.

21.2 Adding a Menu Bar and Toolbar to the Screenset

One of the most important features to note when using class library menus and toolbars is that each toolbar button has an associated menu item. When a toolbar button is selected, a menu event is generated by the class library for its associated menu item. Similarly, when a menu item is enabled, disabled, checked or unchecked, its associated toolbar button is also.

Before using any of the control programs, a common data area must be defined in the screenset Data Block. This is used to pass information between a screenset and the control programs at run time.

21.2.1 Defining the Data Items

Each user control that you define in a screenset must have a Data Block item (master field) associated with it. The data item must be defined as `OBJ-REF` as it is used to hold the class library object reference of the created control. The data item must be defined before you add the user control. If you do not have any object references defined in your screenset, then Dialog System will not let you define a user control.

Include the following data item in the Data Block of the screenset, to hold the object reference of the menu bar and toolbar:

```
MYTOOLBAR          OBJ-REF
```

If you have completed the previous chapter, *Tutorial - Adding and Customizing a Status Bar*, you will not need to include the following data definition for FUNCTION-DATA in the screenset Data Block:

```
FUNCTION-DATA                             1
  WINDOW-HANDLE                  C5     4.0
  OBJECT-REFERENCE               OBJ-REF
  CALL-FUNCTION                  X     30.0
  NUMERIC-VALUE                  C5     4.0
  NUMERIC-VALUE2                 C5     4.0
  SIZE-WIDTH                     C5     4.0
  SIZE-HEIGHT                    C5     4.0
  POSITION-X                     C5     4.0
  POSITION-Y                     C5     4.0
  IO-TEXT-BUFFER                 X    256.0
  IO-TEXT-BUFFER2                X    256.0
```

because FUNCTION-DATA is common to all the control programs, so you need define it only once in each screenset that uses the control programs.

If you do need to include the FUNCTION-DATA definition, you can import it from the file **funcdata.imp** in the **DialogSystem\source** subdirectory. Select **File**, **Import**, **Screenset** and click **OK** to acknowledge the currently loaded screenset might be overwritten. Click the **File** button, double-click **funcdata.imp**, click the **Import** button and then **OK** and **Close**.

Next you need to import the following:

```
TBAR-PARMS                             1
      MENU-INDEX                   9      2.0
      CALLBACK-ENTRY-NAME          X     32.0
      ACCEL-FLAGS                  9      3.0
      ACCEL-KEY                    9      3.0
      MENU-TEXT                    X    256.0
      MENU-HINT-TEXT               X    256.0
      RESOURCE-FILE                X    256.0
      RESOURCE-ID                  9      5.0
      TOOL-TIP-TEXT                X    256.0
      INSERT-BUTTON-BEFORE         9      2.0
      MSG-BOX-TEXT                 X    256.0
```

from the file **tbardata.imp** in the **DialogSystem\source** subdirectory in the same way as described above.

MYTOOLBAR will be used to store the object reference of the created toolbar. This data item will be the masterfield for the toolbar user control defined in the steps below.

TBAR-PARMS is a general purpose group used to pass information about menu items and toolbar buttons to the control program when using the toolbar functions.

To enable the class library, define CONFIG-FLAG and CONFIG-VALUE as C5 4.0 items in your Data Block, and then add the following dialog to the beginning of the SCREENSET-INITIALIZED event in the global dialog of the screenset:

```
CLEAR-CALLOUT-PARAMETERS $NULL
CALLOUT-PARAMETER 1 CONFIG-FLAG $NULL
CALLOUT-PARAMETER 2 CONFIG-VALUE $NULL
MOVE 15 CONFIG-FLAG
MOVE 1 CONFIG-VALUE
CALLOUT "dsrtcfg" 3 $PARMLIST
```

This dialog must be placed before any other dialog in the SCREENSET-INITIALIZED event.

21.2.2 Defining the Menu Bar and Toolbar

When you have defined the required data, define the user control:

1. Select the Customer Details window, MAIN-WINDOW, to which you want to add the menu bar and toolbar.
2. Select **Toolbar** on the **Programmed Controls** toolbar.
3. Position and size the toolbar along the top of the window, where you would expect to see the menu and toolbar.
4. Complete the following items on the Toolbar Properties dialog box:

   a. Choose an appropriate name for the toolbar. We will use **MAIN-WINDOW-TOOLBAR**.

   b. Specify the master field name. It must match the Data Block name you defined earlier, so **MYTOOLBAR**.

   c. Specify **TBAR2** as the name of the toolbar control program.

   d. Check Add program to current project.

   e. Click **Generate** to generate the toolbar control program **tbar2.cbl**.

      A message box appears, asking you if you want to generate a class library menu structure. Click **Yes**.

      A message box appears, reminding you that you need to include the necessary entries in the Data Block for use by the generated control program. You have done this in the earlier steps, so click **OK** to continue.

      You can see the generated program being added to your Customer project in the background.

5. Click **OK** and save the screenset.
6. Select the **Generate copyfile** button on the toolbar, and confirm you want to overwrite **customer.cpb**.
7. Compile the generated control program using **Rebuild All** on the Net Express **Project** menu to obtain an executable version of this program.

21.3 Running Your Screenset

Once you have completed all these steps with your screenset, you can run it to see the User Control working.

At this stage, the menu bar and toolbar will not do anything useful.

You need to add dialog for the menu bar and toolbar to work correctly.

TBAR2 is the name of our customized version of the toolbar control program. This will be our controlling program for our menu.

To enable the customized version of the toolbar control to be called, add the following dialog at the end of the screenset’s global dialog:

```
    CALLOUT-TBAR
 CALLOUT "TBAR2" 0 $NULL
```

If you have not completed the previous chapter, you will need to add a WINDOW-CREATED event in Object Dialog. If you have completed the previous chapter, add the following dialog to the WINDOW-CREATED event:

```
    INVOKE MYTOOLBAR "show" $NULL
```

Save the screenset.

21.4 Defining a Menu Structure

Select **Edit**, **Menu Bar** or the Edit menu bar icon from the toolbar. The Menu Bar Definition dialog box appears.

Because we are going to to use the menu created by the control program, and not the one created by DSGRUN, select **Options**, and check **Use Class Library**. This ensures that DSGRUN does not itself create the menu.

21.4.1 Adding New Menu Options

As an example, we will add two new menu options, **Edit** and **Options,** either side of the existing **Orders** menu option. **Edit** will have one menu item, **Select all**. **Options** will have a menu item, **Customize**, which will open a submenu containing **Fonts**.

1 Click on **Orders**, and select **Edit**, **Insert before**.

2 In the Menu Bar Choice Details dialog box, enter EDIT in the Choice name field, and Edit in the Choice text field. Leave all the other fields as they are, and click **OK**.

  You can see that a new menu item, **Edit,** has been added, with a submenu choice of **Edit**.

3 Click on the **Edit** submenu choice, and select **Edit**, **Change**.

4 In the Pulldown Choice Details dialog box, change the Choice text field to be Select all, and enter "select all" in the Class Library hint text field. Choose F6 from the Shortcut key pulldown. Click **OK**.

5 Click on **Orders** again, as in step 1, and this time, select **Edit**, **Insert after**.

6 In the Menu Bar Choice Details dialog box, enter OPTIONS in the Choice name field, and Options in the Choice text field. Click **OK**.

  You can see that a new menu item, **Options,** has been added, with a submenu choice of **Options**.

7 Click on the **Options** submenu choice, and select **Edit**, **Change**.

8 In the Pulldown Choice Details dialog box, change the Choice text field to be Customize, and enter "customize" in the Class Library hint text field. Click **OK**.

9 Click on **Customize**, and select **Edit**, **Subpulldown**.

  You can see that **Customize** now has a submenu choice of **Customize**.

10 Click on this submenu choice, and select **Edit**, **Change**.

11 In the Pulldown Choice Details dialog box, change the Choice text field to be Fonts, and enter "fonts" in the Class Library hint text field. Choose F9 from the Shortcut key pulldown. Click **OK**.

12 Close the Menu Bar Definition dialog box, and save the screenset.

13 Double-click on the toolbar to open the User Control Properties dialog box. Select Toolbar from the Control type pulldown, and click **Generate**.

A message box appears, asking you if you want to generate a class library menu structure. Click **Yes**.

A second message box appears, asking you if you want to overwrite the entire toolbar program. Click **No** so that only the menu definitions are generated, leaving any toolbar changes intact.

A message box appears, reminding you that you need to include the necessary entries in the Data Block for use by the generated control program. You have done this in the earlier steps, so click **OK** to continue.

You can see the generated program being added to your Customer project in the background.

14 Click **Cancel**.

15 Select **Rebuild All** on the **Project** menu.

16 Run the application using the Net Express **Animate**, **Run** menu choice.

At this stage, you can see the menu options you have added, but they will not do anything if you select them.

21.5 Defining a Toolbar Structure

Look at the data structure defined in **tbar2.cbl** as an example of how to create your own toolbar. To do this, from within **tbar2.cbl**, select the Net Express **Search**, **Locate Definition** menu choice to locate the data item "bData".

21.5.1 The Existing Toolbar Structure

The bData table structure defines an ordered list of buttons to be added to the toolbar. Buttons are added to the toolbar in the order in which they appear in the list.

An example of a single button record is shown below:

```
*>-------------------------------------------------------------
*  Button object reference.
   03 object reference.

*  Menu item index to associate button with or zero if this
*  button is to be a separator. This index refers to an index
*  within the mData table.
   03 pic x comp-5 value 2. *> "Exit" menu item

*  Resource ID of button bitmap.
   03 pic x(4) comp-5 value IDB-EXIT.

*  Tool tip to be displayed when mouse is over button.
   03 pic x(bStringSize) value z"Quit this application".
*>-------------------------------------------------------------
```

Each element of the table defines a button on the toolbar and is made up of four parts:

1. Object reference of the created toolbar button.
2. The index (within mData table) of the menu item to associate the button with (or zero for a separator).
3. Resource ID of a bitmap within a resource file to use for the button.
4. Tool tip to be displayed when mouse is over the button.

The default toolbar structure that has been generated contains a separator and a button associated with menu index 2. However, as we have generated a menu structure from the existing Dialog System menu, the index of the File, Exit menu item is in fact 7. Therefore, to associate the second button with the correct menu index, change it from 2 to 7.

21.5.2 Adding New Toolbar Buttons

As an example, we will add two new toolbar buttons for **Select All** and **Fonts**.

To change the toolbar defined in **tbar2.cbl**, use the existing button records as an example to change the button structure.

1. Duplicate the second element of the bData table structure that represents the Exit button, immediately below the existing one. We will now edit the new, duplicated, code.
2. Change the index of the menu item from 7 to 9. Change the comment as appropriate.
3. Change the resource ID of the button bitmap from IDB-EXIT to IDB-BLUE.
4. Change the tool tip from "Quit" to "Select all".
5. Duplicate the lines of code used for the second toolbar button, immediately below that one.
6. Change the index of the menu item from 9 to 14. Change the comment as appropriate.
7. Change the resource ID of the button bitmap from IDB-BLUE to IDB-RED.
8. Change the tool tip from "Select all" to "Fonts".
9. Select Rebuild All on the Project menu.
10. Run the screenset.

    You can now see the two buttons you have just created displayed below the menu items in the toolbar.

    Again, the toolbar buttons will not do anything if you select them.

21.6 Customizing theToolbar

In the previous section we added an existing bitmap button from the file **tbresid.cpy**. In this section, we will create a new button bitmap.

21.6.1 Setting up Resource Files

Add a new resource file to the project and create bitmap resources the toolbar will use.

1. From a Net Express command prompt, copy the bitmaps (***.bmp** files) and the file **mfres.h** from your **DialogSystem\demo\CommonControls\Toolbar** subdirectory to your **DialogSystem\demo\customer** subdirectory. Copy **tbar.rc** and rename it to **tbres.rc**.
2. Right-click in the project window within Net Express and select **Add file to source pool**.
3. Select **tbres.rc**.
4. Right-click on **tbres.rc** and select **Package selected files**, **Dynamic Link Library (DLL)**, and click **Create** to add it to the left-hand pane.
5. Right-click on **tbres.rc** in the left-hand pane and select **Compile** and then **Edit**.
6. Right-click on "Bitmap" in the tree structure and select **New** from the context menu.
7. Enter "IDB_BUTTON1" in the Resource Identifier field.
8. Enter "selectall.bmp" in the Resource Filename field.
9. Enter 220 in the ID Value field and click **OK**.
10. Select **File**, **New**, **Bitmap** from the Image Editor menu, and click **OK**.
11. Enter 16 for the width and height of the bitmap, and click **OK**.
12. Draw the bitmap to be displayed when the toolbar button is enabled.
13. Save the image as **selectall.bmp** and close Image Editor.
14. Repeat this process with "IDB_BUTTON2", ID value 221 and **custfont.bmp** so that both required bitmap resources (and their IDB_ identifiers) are defined in the resource file.Ip
15. Select **File**, **Save As**, and save this resource script as **tbres.rc**.
16. Close **tbres.rc**.
17. Right-click on **tbar2.cbl** and select **Edit** from the context menu.

18 Change the resourceDllName to be **tbres.dll**.

19 Below the line:

```
copy "tbresid.cpy"
```

add in the new bitmaps you have created by entering:

```
78 IDB-Button1             value 220.
78 IDB-Button2             value 221.
```

20 Save **tbar2.cbl** and close it.

21 Select **Update All Dependencies** from the **Project** menu.

22 Rebuild the project. This will create **mfres.cpy** which contains your resource identifier constants needed in **tbar2.cbl**. For now just ignore any errors that you might receive on Rebuild.

21.6.2 Setting up Copyfiles

Make copyfile changes to define the toolbar structure.

We are going to review the toolbar structure in **tbar2.cbl** which provides a menu item and associated toolbar button. This file contains data structures that define how the toolbar will look and respond to events.

We will look at how to define the toolbar.

1 Right-click **tbar2.cbl** in the left-hand pane and select **Edit**.

2 Select the Net Express **Search**, **Locate Definition** menu choice to locate the data item "bData.".

3 Note the bData group structure which defines each toolbar item as described in the section *Defining a Toolbar Structure* earlier in this chapter.

Ensure that the button definition data record contains a reference to an IDB_ resource identifier you defined in the resource file earlier. That is, IDB-Button1 and IDB-Button2.

Save **tbar2.cbl** and close it.

21.6.3 Adding Dialog to the Screenset

Add dialog to the screenset to respond to a menu selection as follows.

If you are adding a toolbar to an existing application (which in this instance we are):

1. Add the following dialog to the toolbar's parent window:

   ```
   USER-EVENT
       XIF= $EVENT-DATA 37000 @EXISTING-DIALOG-MENU-PROCEDURE
   ```

2. Replace the executed procedure with your own menu dialog table. In this example, we will use TEST-MENU-CHOICE, so enter:

   ```
   USER-EVENT
     XIF= $EVENT-DATA 37000 TEST-MENU-CHOICE
   ```

3. Add the following dialog immediately below the above lines:

   ```
   TEST-MENU-CHOICE
     IF= NUMERIC-VALUE(1) 7 @EXIT
     IF= NUMERIC-VALUE(1) 9 SELECT-ALL
     IF= NUMERIC-VALUE(1) 14 FONTS
   ```

4. At the end of the dialog, add the following lines:

   ```
   SELECT-ALL
     BEEP
   FONTS
     BEEP
     BEEP
   ```

   If you are adding class library menu items to replace existing menu items, you can simply branch to the procedure name (prefixed with the @ symbol) that was previously used to handle the menu event. Alternatively, you could make a COBOL CALL to a subprogram from your toolbar callback code following the class library menu event.

If you are developing a new application:

1. Add the following dialog to the USER-EVENT dialog for the toolbar's parent window:

   ```
   USER-EVENT
       XIF= $EVENT-DATA 37000 DIALOG-PROCEDURE

   DIALOG-PROCEDURE
   ```

```
*   Dialog function processing - perhaps show a message box
```

As an alternative to the use of dialog functions to handle the class library menu event, your toolbar callback code could simply make a COBOL CALL to an existing program to perform that menu function.

Save the screenset, switch to the Net Express IDE and rebuild the project.

You can now select **Animate**, **Run**, to test your screenset. When you select either **Select All** from the **Edit** menu, or the toolbar button, the dialog procedure that you coded will be executed. Try this for **Fonts**, from the **Options**, **Customize** menu too.

22 Tutorial - Adding an ActiveX Control

This chapter describes how to add an ActiveX Control to your screenset. You need to have read the chapter *Programming Your Own Controls* and the previous tutorials before looking at this one.

The following sections take you step by step through changing a screenset to use the supplied clock ActiveX control.

22.1 Screenset Alterations

The aim of this tutorial is to display the supplied ActiveX clock control when you double-click on the time on the status bar. You will be shown how to use an ActiveX control later, but for now we will open **customer.gs** in Net Express and define the following objects in the screenset:

- A dialog box to hold the ActiveX clock.
- A dialog box that will allow the alarm time to be set.
- A message box to display a message when the alarm goes off.

1. Make the dialog box to hold the ActiveX clock about a third of the size of the customer screenset, and place it anywhere.
2. Name this object CLOCK-DIALOG.
3. Leave all its properties as default.
4. Define the parent as being $WINDOW.
5. Create a message box to display a message when the alarm goes off.
6. Name this object ALARM-GONE-OFF-MBOX.

7 Leave all its properties as default.

8 Define the parent as being $WINDOW.

9 Add two PIC 9(2) data items, ALARM-HOURS and ALARM-MINUTES, to the Data Block for this screenset.

10 Make the dialog box that will allow the alarm time to be set about a third of the size of the customer screenset, and place it anywhere.

11 Name this object SET-ALARM-DIALOG.

12 Leave all its properties as default.

13 Define the parent as being $WINDOW.

14 On the CLOCK-DIALOG dialog box:

a Create a push button with its text set to "Set Alarm".

b Add the following dialog to the Set Alarm button:

```
BUTTON-SELECTED
    MOVE " " IO-TEXT-BUFFER2(1)
    SET-FOCUS SET-ALARM-DIALOG
```

c Create a push button with its text set to "Close".

d Add the following dialog to the Close button so that it unshows the CLOCK-DIALOG dialog box when it is clicked.

```
BUTTON-SELECTED
    UNSHOW-WINDOW CLOCK-DIALOG $NULL
```

Do not delete the CLOCK-DIALOG dialog box, as this will cause the alarm not to function.

15 On the SET-ALARM-DIALOG:

a Create an entry field to hold the hour of the alarm time.

Link this to the ALARM-HOURS master field.

b Create an entry field to hold the minutes of the alarm time.

Link this to the ALARM-MINUTES master field.

c Create an entry field to hold the message to be displayed when the alarm is triggered.

The master field for this entry field should be IO-TEXT-BUFFER2.

d Create a push button with its text set to "OK". The **OK** button should have the following dialog:

```
BUTTON-SELECTED
    BRANCH-TO-PROCEDURE SET-ALARM
```

e Create a push button with its text set to "Cancel". The **Cancel** button should have the following dialog to enable the SET-ALARM-DIALOG dialog box to be deleted when the user clicks **Cancel**:

```
BUTTON-SELECTED
    DELETE-WINDOW SET-ALARM-DIALOG $NULL
```

f Add the following dialog to the SET-ALARM-DIALOG dialog box itself:

```
 CR
     BRANCH-TO-PROCEDURE SET-ALARM

 ESC
     DELETE-WINDOW SET-ALARM-DIALOG $NULL

   SET-ALARM
* Dialog will be added here later.
```

Having completed these steps, you can add the dialog to use the left-mouse-button-double-click event that you added in the earlier tutorial to display the CLOCK-DIALOG dialog box. See the section *Customizing the Status Bar Control Program* in the chapter *Tutorial - Adding and Customizing a Status Bar*.

In the left-mouse-button-double-click event callback, you added code to send user event 34591 when the callback was triggered. The code you added also places the section number of the status bar on which the double-click event happened into the NUMERIC-VALUE(1) data item.

With this information, you can add the following to the MAIN-WINDOW dialog table to process the double-click user event:

```
USER-EVENT
    XIF= $EVENT-DATA 34591 DOUBLE-CLICK-EVENT

  DOUBLE-CLICK-EVENT
    IF= NUMERIC-VALUE(1) 5 DOUBLE-CLICK-ON-CLOCK

  DOUBLE-CLICK-ON-CLOCK
    SET-FOCUS CLOCK-DIALOG
```

This dialog causes the CLOCK-DIALOG dialog box to be shown if the MAIN-WINDOW window receives user event number 34591, and NUMERIC-VALUE(1) is set to 5 (the 5th section of the status bar contains the clock).

22.2 Using the Dialog System Clock ActiveX Control

A separate demonstration of applying this functionality is provided in the **clock.gs** screenset and documented in the file **clockds.txt**

If you have followed the tutorial, you will have created a CLOCK-DIALOG dialog box in your screenset. You can now add the supplied Dialog System clock ActiveX control to this dialog box.

Add the following data definitions to your screenset:

```
CLOCK-DIALOG-ACTIVEX-OBJREF          OBJ-REF
ACTIVEX-PARAMETERS                         1
  PARM-NAME                      X      30.0
  P1                             C5      4.0
  P2                             C5      4.0
  P3                             OBJ-REF
```

You must make sure that the last four items are indented, to show they are group items.

The first data item listed is used to store the object reference of the ActiveX . The master field group specified is used as the second parameter when calling the ActiveX control program.

The next step is to position the ActiveX control on the CLOCK-DIALOG dialog box:

1 Select **File**, **Import**, **ActiveX control...**.

2 Select Dialog System Clock.

   The control is added to the ActiveX toolbar, from where you can select it and place it on the status bar of the customer screenset.

The ActiveX Control Properties dialog box opens when you position the ActiveX control.

3 Specify a name for the ActiveX control, set the master field to be CLOCK-DIALOG-ACTIVEX-OBJREF, and set the program name to be **CLOCKCTRL**.

4 Click **Generate** to generate the ActiveX control program.

Click **OK**.

5 You now need to set the properties for the ActiveX control. Each ActiveX control has tailorable properties which determine the run-time behavior of that control.

When the control has been painted, a Property List dialog box is displayed. This shows some (not always all) of the configurable properties that the control provides. You can change the values in either this list or use the Controls context menu to view the full list of properties via the ActiveX property page context menu choice.

In the case of the Dialog System clock ActiveX, the available properties are the background bitmap, and whether the clock displays as analog or digital.

You need to add the dialog to handle the user event that indicates that the alarm time has been reached. Add the following to the CLOCK-DIALOG dialog box:

```
USER-EVENT
    IF= $EVENT-DATA 34570 ALARMGONEOFF

  ALARMGONEOFF
    INVOKE-MESSAGE-BOX ALARM-GONE-OFF-MBOX
                               IO-TEXT-BUFFER2(1) $REGISTER
```

Finally, you need to complete the SET-ALARM procedure that you added to the dialog for the SET-ALARM-DIALOG dialog box earlier. This procedure invokes the SetAlarm method in the ActiveX control to set the alarm. Add the following dialog below SET-ALARM:

```
    SET-MOUSE-SHAPE SET-ALARM-DIALOG "SYS-WAIT"
    MOVE ALARM-HOURS P1(1)
    MOVE ALARM-MINUTES P2(1)

    NULL-TERMINATE IO-TEXT-BUFFER2(1)
    CLEAR-CALLOUT-PARAMETERS $NULL
    CALLOUT-PARAMETER 1 IO-TEXT-BUFFER2(1) $null
    CALLOUT-PARAMETER 8 P3(1) $NULL
    INVOKE "chararry" "withValue" $PARMLIST
```

This dialog demonstrates the passing of the Message text (in IO-TEXT-BUFFER2(1)) as an object reference to a CharacterArray instance. Note the use of the functions NULL-TERMINATE and INVOKE for the P3(1) variable. These functions are described in detail in the Help. Add the following dialog:

```
MOVE "INVOKE-ActiveX-METHOD" CALL-FUNCTION(1)
    SET OBJECT-REFERENCE(1) CLOCK-DIALOG-ActiveX-OBJREF
    MOVE "SetAlarm" PARM-NAME(1)

    CLEAR-CALLOUT-PARAMETERS $NULL
    CALLOUT-PARAMETER 1 FUNCTION-DATA $NULL
    CALLOUT-PARAMETER 2 ActiveX-PARAMETERS $NULL
    CALLOUT "ocxctrl" 0 $PARMLIST

    CLEAR-CALLOUT-PARAMETERS $NULL
    CALLOUT-PARAMETER 8 P3(1) $NULL
    INVOKE P3(1) "finalize" $PARMLIST

    DELETE-WINDOW SET-ALARM-DIALOG $NULL
```

You have now completed adding the Dialog System clock ActiveX control to your screenset. Save your screenset and run it to try out the changes you have made.

23 Tutorial - Using Bitmaps to Change the Mouse Pointer

The chapters *Window Objects* and *Control Objects* described how to choose bitmap graphics. This chapter shows you how to:

- Change the mouse pointer.
- Provide functionality for your bitmaps.

23.1 Changing the Mouse Pointer

To change the shape of the mouse pointer, use the SET-MOUSE-SHAPE function. For example, you might want to change the mouse pointer to an hour glass or a clock to inform your user to wait until an operation is complete.

See the topic *Dialog Statements: Functions* in the Help for more information on the SET-MOUSE-SHAPE function.

23.1.1 The Moudemo Sample Screenset

The sample program **Moudemo** (available on your samples disk) changes the mouse pointer as it passes over different objects.

- When the static pointer radio button is selected, movement of the mouse over the three objects (entry field, list and bitmap) has no effect.
- When the dynamic pointer radio button is selected, the mouse pointer changes shape as it passes over different objects.

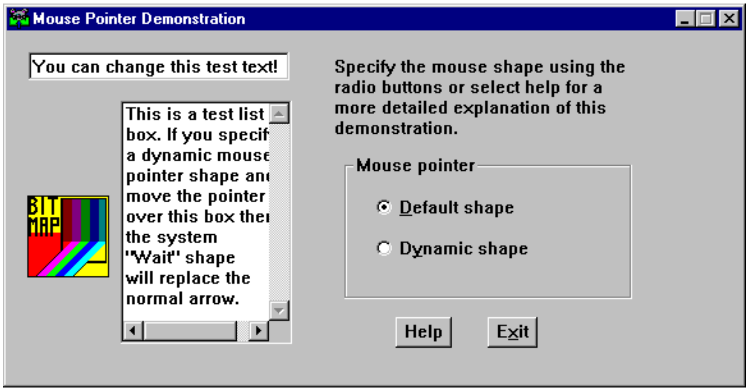
There is also an entry field used for text entry where the cursor shape changes to a text bar and a list box where each line can be selected and highlighted.

To test the Moudemo screenset you can:

1. Start Dialog System.
2. Load the screenset **moudemo.gs**.
3. Select **Run** on the **File** menu to invoke the Screenset Animator.
4. Accept the default Screenset Animator values.
5. Press **Enter**.

   You see the screen shown in Figure 23-1.

Figure 23-1. The Moudemo Screen



Mouse Pointer Demonstration

You can change this test text!

This is a test list box. If you specif a dynamic mouse pointer shape an move the pointer over this box the the system "Wait" shape will replace the normal arrow.

Specify the mouse shape using the radio buttons or select help for a more detailed explanation of this demonstration.

Mouse pointer

● Default shape

○ Dynamic shape

Global dialog is used to return control to the calling program if the escape key is pressed or the main window is closed:

```
ESC
    RETC
CLOSED-WINDOW
    RETC
```

When the screenset is initialized the static radio button is set on, and a text string is placed into the entry field. The default mouse pointer shape is set to `SYS-Arrow`:

```
SCREENSET-INITIALIZED
    SET-BUTTON-STATE DEFAULT-RB 1
    MOVE "You can change this test text!" TEST-TEXT-DATA
    REFRESH-OBJECT TESTTEXT-EFLD
    SET-MOUSE-SHAPE MOUSEDEMO-WIN "SYS-Arrow"
```

If the **Exit** button is selected, the Exit message box is displayed and the value of $REGISTER is checked. A value of 1 in $REGISTER indicates **OK** was selected, meaning the user wants to exit:

```
BUTTON-SELECTED
    INVOKE-MESSAGE-BOX EXIT-MSG $NULL $REGISTER
    IF= $REGISTER 1 EXITAPP
  EXITAPP
     RETC
```

Additional dialog is attached to each object as required. When the static radio button is selected (default at screenset initialization), the `SYS-Arrow` shape is used over the whole of the main window:

```
BUTTON-SELECTED
    SET-MOUSE-SHAPE MOUSEDEMO-WIN "SYS-Arrow"
```

When the dynamic radio button is selected, the shape of the mouse pointer depends on the object it is positioned over. `SYS-Move` are provided by the operating environment and `pencil-ptr` has been created especially for this screenset:

```
BUTTON-SELECTED
    SET-MOUSE-SHAPE TESTTEXT-EFLD "SYS-Move"
    SET-MOUSE-SHAPE TESTMAP-BMP "pencil-ptr"
```

If the **Help** button is selected, the help dialog is displayed:

```
BUTTON-SELECTED
    SET-FOCUS HELP-DIAG
```

Clicking **OK** in the Help window removes the window from the display and returns control to the main window:

```
BUTTON-SELECTED
    DELETE-WINDOW HELP-DIAG MOUSEDEMO-WIN
```

There is no dialog attached to the list box or entry field.

23.1.1.1 Changing the Side File

If you installed the samples, a resource side file was loaded for the moudemo sample (**moudemo.icn**). Therefore you do not need to edit the **ds.icn** file to add the mouse pointer.

However, this section describes how to add the mouse pointer and bitmap to the standard **ds.icn** file.

The mouse pointer `pencil-ptr` has been created for use in this screenset and placed in a DLL file. Therefore, you must make an entry in the **ds.icn** file under the appropriate section heading:

```
pencil-ptr        : dssamw32.dll 002
```

Mouse pointers must be placed in a DLL file. The procedure for doing this is described in the topic *Bitmaps, Icons and Mouse Pointers* in the Help.

You must also add an entry for the new bitmap stating the name of the bitmap and its location. It does not have to be in the same directory as the screenset. In the Moudemo sample screenset this bitmap is called **colorful.bmp**. The corresponding entry in **ds.icn** (under the appropriate section heading) would be:

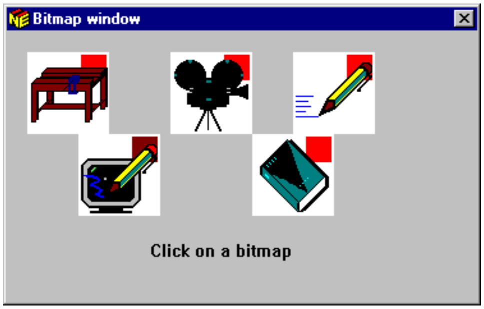
```
colorful : dssamw32.dll 003
```

23.2 Programming Bitmaps

In Dialog System, a bitmap is a control that you can select. In this respect, a bitmap is like a radio button. What happens after the bitmap is selected depends on how you code the dialog.

Because a bitmap is represented by a graphic image, it can be more meaningful to users. Figure 23-2 shows a window with several bitmaps attached to it.

Figure 23-2. Window with Bitmaps Attached



To help you understand bitmaps, the following example contains five bitmaps representing different functions the user can perform. When the user selects a bitmap, return to the calling program and invoke the function represented by the bitmap.

These bitmap controls are named WBBMP, ANIMBMP, HELPBMP, SCREENBMP and EDITBMP. Each bitmap has a handle, that is, an internal identifier. The handle that is assigned to an object must be defined in the Data Block.

Handles for this example (and in fact all handles) are defined as:

```
WB-HANDLE                C         4.00
ANIM-HANDLE              C         4.00
HELP-HANDLE              C         4.00
SCREEN-HANDLE            C         4.00
EDIT-HANDLE              C         4.00
```

The first thing you must do is save the handles of the bitmaps. The best way to do this is in global dialog using the SCREENSET-INITIALIZED function.

```
SCREENSET-INITIALIZED
    MOVE-OBJECT-HANDLE ANIMBMP ANIM-HANDLE
    MOVE-OBJECT-HANDLE EDITBMP EDIT-HANDLE
    MOVE-OBJECT-HANDLE WBBMP WB-HANDLE
    MOVE-OBJECT-HANDLE HELPBMP HELP-HANDLE
    MOVE-OBJECT-HANDLE SCREENBMP SCREEN-HANDLE
```

The `MOVE-OBJECT-HANDLE` function moves the handle for the bitmap (`ANIMBMP`) to a numeric field (`ANIM-HANDLE`).

BITMAP-EVENT is the only event associated with a bitmap. When the user clicks on a bitmap, the event is triggered and the handle of the bitmap is stored in $EVENT-DATA.

This dialog is attached to the window on which the bitmaps appear:

```
BITMAP-EVENT
    IF= $EVENT-DATA ANIM-HANDLE SET-ANIM
    IF= $EVENT-DATA EDIT-HANDLE SET-EDIT
    IF= $EVENT-DATA WB-HANDLE SET-WB
    IF= $EVENT-DATA HELP-HANDLE SET-HELP
    IF= $EVENT-DATA SCREEN-HANDLE SET-SCREEN
  SET-ANIM
    MOVE 1 FUNCTION
    RETC
  SET-EDIT
    MOVE 2 FUNCTION
    RETC
  SET-WB
    MOVE 3 FUNCTION
    RETC
  SET-HELP
    MOVE 4 FUNCTION
    RETC
  SET-SCREEN
    MOVE 5 FUNCTION
    RETC
```

Thus, when the user clicks on a bitmap:

- The bitmap event is triggered and the handle of the bitmap is stored in $EVENT-DATA.
- $EVENT-DATA is compared to the handles of the bitmaps that were stored earlier.
- A procedure is performed that sets a value in `FUNCTION`.
- Control is returned to the program .

Now your program executes the appropriate function based on the value in `FUNCTION`.

For more information on adding bitmaps to your Dialog System interface, see the topic *Bitmaps, Icons and Mouse Pointers* in the Help.

A Fonts and Colors

This appendix looks at how you can use different fonts and colors. You might want to change the appearance of your interface to:

- Conform to your organization's standards. For example, your organization might have a particular color scheme that it uses for all applications.
- Suit your own personal preference.
- Draw the user's attention to a particular component. For example, you can:
    - Highlight a default value with a different font.
    - Change the background color of a field that has caused a validation error.
    - Use a bold typeface to emphasize a **Delete** push button.

Using different fonts and colors can greatly enhance your interface. However, using too many different fonts can make a screen look messy and it is advisable to use no more than three colors on any screen.

A.1 Setting Fonts

A font is a set of characters having common visual characteristics. You can use a different font to give the interface a more distinctive look. For example, you can use a monospaced font to ensure that parts of the interface line up neatly.

You can also use a different font to draw the user's attention to a specific part of the interface. For example, you can highlight a parameter in a text string by making the parameter a different font from the surrounding text. The following text illustrates this feature:

In the dialog fragment, `large-entry-field` is the data item associated with the MLE.

The size of the text (its pointsize) is another way to highlight a specific component of the interface. In the preceding example, you could make the type used for the filename a larger pointsize, for example 12 point instead of 10 point. This would also draw the user's attention to the filename.

You can set fonts at definition time only. No Dialog System functions are available to change any of the font definitions. However, you can dynamically change fonts using Panels V2. The chapter *Using Panels V2* describes how to call Panels V2 from your Dialog System program.

See the topic *Dialog System Overview* in the Help for a description of setting fonts.

A.2 Setting Colors

You can change the foreground and background colors of objects either at definition time, using the **Color** menu choice on the **Edit** menu or clicking on the edit color toolbar icon, or dynamically, using the SET-COLOR function.

For example, the following dialog fragment changes the color of a field with validation errors to 'WHITE' (foreground color) on 'RED' (background color). Remember that when Dialog System detects a validation error, the identifier of the field in error is placed in $EVENT-DATA.

```
VAL-ERROR
    SET-COLOR $EVENT-DATA 'WHITE' 'RED'
    SET-FOCUS $EVENT-DATA
```

Generic colors are available for all environments. Also, each environment has a list of additional colors that you can select. If you are developing cross-platform applications, check to make sure the colors you are using are supported on all environments.

To set the color of an object back to its default values, use the value of $NULL for the foreground and background colors:

```
SET-COLOR DELETE-PB $NULL $NULL
```

For further information on the SET-COLOR function and lists of available colors, see the topic *Dialog Statements: Functions* in the Help.

Index