northgate
INFORMATION SOLUTIONS

UIMS

Version 2.0

DATA/BASIC API Programmer's Guide

Document No. UM70006279B
September 1994

Northgate Information Solutions UK Limited
Boundary Way
Hemel Hempstead, Hertfordshire,
HP2 7HU

Tel: +44 (0)1442 232424
Fax: +44 (0)1442 256454

www.northgate-is.com

Comment Sheet

Please give page number and description for any errors found:

Page	Error

Please use the box below to: describe any material you think is missing; describe any material which is hard to understand; enter any suggestions for improvement; provide any specific examples of how you use your system which you think might be useful to readers of this manual

Continue on a separate sheet if necessary.

If you would like someone to contact you about documentation, please tick this box. ☐

Important: *Please enter your name, address and telephone number on the back of this form before returning.*

Manual: UIMS DATA/BASIC API, Programmer's Guide, UM70006279B, September 1994 .

Name of Sender: ______________________________

Company Name / Address: ______________________

____

____

Telephone: ___________________________________

Date: ____

FOLD B

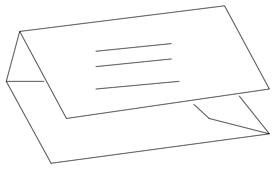
BUSINESS REPLY SERVICE
Licence No. HH698

2

F
O
L
D

A

Technical Publications Department
Northgate Information Solutions UK Limited
Boundary Way
Hemel Hempstead
Hertfordshire HP2 7HU



Fold along fold A, then B and C. Tuck in so name and address are on outside.

FOLD C

Appendix A NewView Examples

List of Figures

Chapter 1
About this Manual

This chapter describes the different sections of the manual and any conventions used.

Purpose of this Manual

This manual is intended to help experienced DATA/BASIC programmers make the transition to writing applications that use the REALITY User Interface Management System (UIMS). It shows you how to provide existing applications with a graphical user interface and explains the basics of writing new UIMS applications. It also explains how to use UIMS subroutines and messages to carry out useful tasks common to all UIMS applications. These explanations are illustrated with example applications that you can compile and run with RealLink for Windows.

It is assumed that, in addition to being an experienced DATA/BASIC programmer, you will be familiar with RealLink for Windows and Microsoft Windows, and have access to the appropriate user manuals. You will also need a copy of the *UIMS DATA/BASIC API Reference Manual*.

This manual consists of the following sections:

Chapter 1, About this Manual, describes the different sections of the manual and any conventions used.

Chapter 2, UIMS Concepts, describes the features that the UIMS environment offers, and compares UIMS applications with standard REALITY applications.

Chapter 3, Converting Existing Applications – NewView, explains how to give an existing character-based application a graphical user interface, with only minimal changes to the source code.

Chapter 4, A Generic UIMS Application, explains how to create a simple UIMS application, which you can use as the basis for your own programs.

Chapter 5, Windows, describes how to create a window, and how to control its appearance and its position on the screen. It also shows you how to display text and graphics in a window.

Chapter 6, Keyboard and Mouse Input, describes the input messages that UIMS sends to your application and shows you how to respond to these messages.

Chapter 7, The Pointer, the Mouse and the Keyboard, describes how to display and the pointer and control its shape, and how to let the user use the keyboard to move the pointer.

Chapter 8, Menus, shows you how to use menus in your application.

Chapter 9, Controls, shows you how to create and use controls for interaction with the user.

Chapter 10, Dialogue Boxes, shows you how to create and use dialog boxes containing controls.

Chapter 11, The Clipboard, describes how to copy data to and paste data into your application from the Windows clipboard. It also describes how to control the Cut, Copy and Paste items on the Edit menu.

Chapter 12, Fonts, shows you how you can use fonts to improve the look of your application.

Chapter 13, Dynamic Data Exchange, describes how to use DDE to extract information from other Windows applications, to automatically update them with new information, and to send commands and keystrokes to manipulate them by remote control.

Chapter 14, Hybrid Applications, describes how to enhance existing applications by creating UIMS modules with their own event loops.

Appendix A, NewView Examples, gives the source code listings for the example programs described in Chapter 3.

Related Documents

UIMS DATA/BASIC API, Reference Manual.

UIMS DATA/BASIC API, Quick Reference Guide.

REALITY DATA/BASIC Reference Manual.

Systems Application Architecture, Common User Access: Advanced Interface Design Guide (IBM).

RealLink for Windows User Manual.

Microsoft Windows User's Guide.

Using the Example Applications

Most chapters in this guide conclude with an example application that illustrates the UIMS features described in the chapter. These applications are written in REALITY DATA/BASIC and conform to the user-interface style recommended in the IBM *Systems Application Architecture, Common User Access: Advanced Interface Design Guide*.

The source files for all example applications are in the file UIMS-EXAMPLES, supplied with the UIMS API. In most cases, the application consists of a DATA/BASIC source item, a resource script, and a header item that is common to both. Compiled versions of the DATA/BASIC code and the resources are also provided. Before running the application, you must do one of the following –

Either:

- Copy the compiled resources to the RealLink resource directory on your PC (this directory is specified in your RFW.INI file).
- Catalog the compiled DATA/BASIC program.

Or:

- Copy the resource script and the header item to your PC and compile the resources using the UIMS Resource Compiler (RLRC). Then copy the resulting resource file to your RealLink resource directory.
- Compile the DATA/BASIC source and then catalog the resulting program.

It is a good idea, when reading the description of an example application, to refer to the corresponding source code. You can also use the sources as the basis for your own applications.

Conventions

This manual uses the following conventions:

`Text`	Bold text shown in this typeface is used to indicate input which must be typed on the keyboard.
`Text`	Text shown in this typeface is used to show text that is output to the screen.
Bold text	Bold text in syntax descriptions represents characters typed exactly as shown. For example: **WHO**
Text	Characters or words in italics indicate parameters which must be supplied by the user. For example, in **GetChildFocus**(*Context*, *Contact*, *vChild*) the parameters *Context*, *Contact* and *vChild* are italicized to indicate that this is the general form of the **GetChildFocus** subroutine. In an actual program, the user supplies particular arguments for the place-holders *Context*, *Contact* and *vChild*. Italic text is also used for titles of documents referred to by this document.
vText	A lower case '*v*' prefixing a parameter name indicates that a variable must be supplied so that a value can be returned. In the above example, for instance, the '*v*' prefix to the parameter name *vChild* indicates that, in an actual program, the user must supply the name of a variable in which to return the handle of the child which currently has the focus.
aText, *vaText*	A lower case '*a*' prefixing a parameter name indicates that either the programmer must supply a dynamic array or, when combined with a lower case '*v*', that on return the parameter will contain a dynamic array with one value in each attribute.

[Brackets]

Brackets enclose optional parameters. For example, in

```
#IFDEF ident
    source code block
[#ELSE
    source code block]
#ENDIF
```

the keyword **#ELSE** and an associated *source code block* can optionally be included.

…

In syntax descriptions, ellipses following a group of items indicate that the parameters preceding can be repeated as many times as necessary. For example, in

```
    ATTRIBUTE = Value
    [ATTRIBUTE = Value
    …]
```

the ellipses indicate that the sequence *ATTRIBUTE = Value* may be repeated as many times as necessary.

In the DATA/BASIC program examples, ellipses are used to split lines of code for clarity. Since in DATA/BASIC an ellipsis indicates that a line of code continues on the next line, this does not prevent compilation of the examples.

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted.

SMALL CAPITALS

Small capitals are used for the names of keys such as RETURN.

CTRL+X

Two (or more) key names joined by a plus sign (+) indicate a combination of keys, where the first key(s) must be held down while the second (or last) is pressed. For example, CTRL+X indicates that the CTRL key must be held down while the X key is pressed.

Enter

To enter text means to type text then press RETURN. For instance, 'Enter the WHO command' means type `WHO`, then press RETURN.

	In general, the RETURN key (shown as ENTER or ↵ on some keyboards) must be used to complete all terminal input unless otherwise specified.
Press	Press single key or key combination but do not press RETURN afterwards.
X'*nn*'	This denotes a hexadecimal value.
REALITY	Throughout this manual, REALITY should be taken to include RealityX unless otherwise stated.
You	This manual is written for the application developer. Depending on the context, the word "you" can refer either to you as a developer, or to the application you are developing.
User	The "user" refers not to you, the application developer, but to the person who will ultimately use the application you write.

Comment Sheet

A Comment Sheet is included at the front of this manual. If you find any errors or have any suggestions for improvements in the manual please complete and return the form. If it has already been used then send your comments to the Technical Publications Manager at the address on the title page.

Chapter 2
UIMS Concepts

The REALITY User Interface Management System adds many features to the standard REALITY environment. Because of this, UIMS applications are, in some ways, more complex than standard REALITY DATA/BASIC programs.

This chapter covers the following topics:

- The relationship between RealLink for Windows and UIMS.
- A comparison of UIMS applications and standard REALITY DATA/BASIC applications.
- Features that the UIMS environment offers, and the impact these features have on the way you develop and write applications.
- The UIMS programming model.
- The process you use to develop UIMS applications.

Introduction

RealLink for Windows is a PC terminal emulator that runs in the Microsoft Windows environment. At the heart of RealLink is a User Interface Manager that provides its interface to the Windows environment and generates its graphical display.

RealLink makes many of the features of the User Interface Manager available to host applications by means of commands that can be transmitted across a LAN or other communications link. The UIMS DATA/BASIC API provides the REALITY DATA/BASIC programmer with a suite of subroutines that can be used in applications. These subroutines simplify the programmer's task by constructing the User Interface Manager commands and transmitting them to RealLink. RealLink, in turn, carries out these commands and returns any results to the host application via variables supplied by the DATA/BASIC programmer.

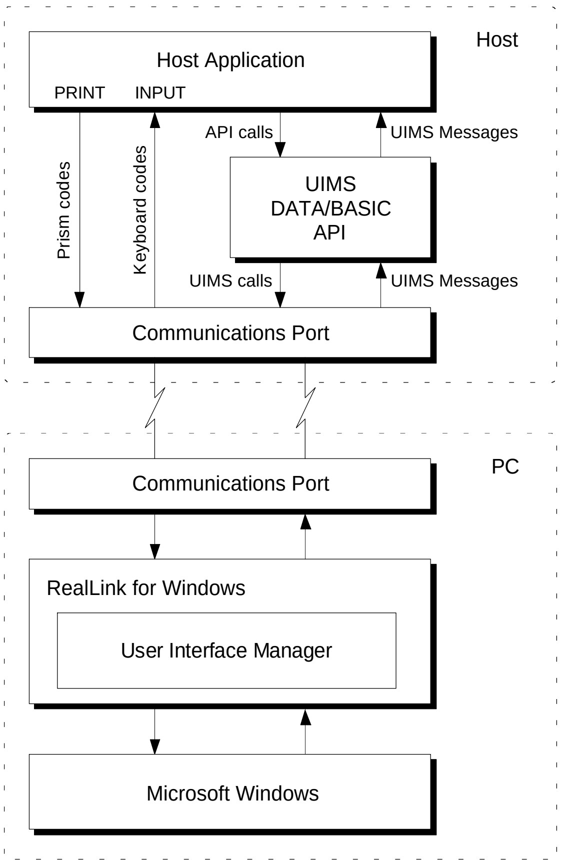


Figure 2-1. The User Interface Management System

UIMS and REALITY: a Comparison

UIMS has many features that the standard REALITY environment does not. For this reason, UIMS applications may, at first, seem more complex than standard REALITY programs. This is understandable when you consider some of the additional features that UIMS offers. These include:

- A graphical user interface featuring windows, menus, dialog boxes and controls for applications.
- Queued input.
- Multitasking.
- Data interchange between applications.

When writing applications for the REALITY environment, programmers use the standard DATA/BASIC statements to carry out a program's input, output and other activities. DATA/BASIC assumes a standard user environment which includes a character-based terminal for user input and output. In UIMS, this assumption is no longer valid. A UIMS application is a collaboration between the user's PC and the remote host, with the application sharing the PC's resources, including the CPU, with other applications. UIMS applications interact with the user through a graphics-based display, a keyboard and a mouse.

The following sections describe some of the major differences between standard REALITY applications and UIMS applications.

The User Interface

RealLink for Windows is an application that runs in the Microsoft Windows environment. Since one of the principal design goals of Windows was to provide the user with visual access to most, if not all, applications at the same time, UIMS applications must share the display with other Windows applications. Some systems do this by giving one program exclusive use of the display, while other programs wait in the background. Windows, however, can give every application access to some part of the display at all times.

An application shares the display with other applications by using a "window" for interaction with the user. Technically, a window is little more than a rectangular portion of the display that Windows allows the application to use. In practice, however, a window is a combination of useful visual devices, such as menus, controls and scroll bars, that the user uses to direct the actions of the application.

In REALITY, the system automatically prepares the display for your application. The DATA/BASIC input and output commands simply assume that this has been done. In UIMS, your application must create its own window before performing any output or receiving any input. Once a window has been created, UIMS provides a great deal of information about what the user is doing with that window. UIMS also automatically carries out many of the tasks that the user requests, such as moving the window and changing its size.

Another advantage offered by the UIMS environment is that, in contrast with a standard REALITY program which has access to a single screen "surface", a UIMS application can create and use any number of overlapping windows to display information. UIMS manages the screen for you and controls the placement and display of windows, ensuring that applications do not attempt to access any part of the display at the same time.

Queued Input

One of the biggest differences between UIMS applications and standard DATA/BASIC programs is the way they receive user input.

In the normal REALITY environment, a program reads from the keyboard by making an explicit call to the INPUT DATA/BASIC statement. This statement typically waits until the user presses RETURN before returning a string of characters to the program. In contrast, in the UIMS environment, UIMS receives all input from the keyboard, mouse and timer and places the input in the application's "message queue". When the application is ready to receive input it simply reads the next message from its message queue.

In the normal REALITY environment, input is typically in the form of 7- or 8-bit characters from the keyboard. The INPUT statement reads characters from the keyboard and returns ASCII codes corresponding to the keys pressed.

In UIMS, an application receives input in the form of "input messages" that UIMS sends it. A UIMS input message contains information that exceeds the type of input information available in the standard REALITY environment. Depending on the source of the message, it can specify a code representing the key (if a key has been pressed), the states of the SHIFT, CTRL and ALT keys, the position of the mouse, the mouse button that has been pressed or released, and the system time. For example, there are two mouse messages, UIMS.MSG.PRESS and UIMS.MSG.RELEASE, that correspond to the press and release of a mouse button. With each mouse message, UIMS provides the position of the mouse, and the states of all the mouse buttons and of various keys on the keyboard (such as SHIFT, CTRL and NUMLOCK), in addition to a code that identifies the button. Other mouse messages have the same format and are processed in the same way.

Multitasking

Since RealLink for Windows (and thus UIMS) runs in the Microsoft Windows environment, it can take full advantage of Windows' multitasking capabilities. The user can run a number of different applications at the same time, or in many cases two or more instances of the same application, and swap between them as required. On REALITY, the user is restricted to running only one application at a time, but by using two or more instances of RealLink, the user can run multiple REALITY and UIMS applications.

UIMS also provides access to the Windows clipboard, so that users can transfer data between applications.

Shared Processing

Within your host application, you make use of UIMS by calling the cataloged subroutines that form the UIMS DATA/BASIC API. The subroutines do not, however, call UIMS directly, but rather construct "UIMS calls" that are transmitted over the communications link to the PC. On receipt of a UIMS call, RealLink calls the appropriate UIMS function on the PC. UIMS, in turn, calls Windows functions to manage the user interface.

This means that the task of running your application is shared by the host and the PC. The PC takes charge of the user interface, allowing the host to concentrate on data processing.

In addition, you can store definitions of your graphical resources (windows, menus, controls, etc.) on the PC and simply instruct UIMS to load them when required. Because the load operation is carried out entirely on the PC, the communication needed between the host and the PC is reduced and the speed of the application is increased.

The UIMS Programming Model

Most UIMS applications use the following elements to interact with the user:

- Windows
- Menus
- The message loop

This section describes these elements in detail.

Windows

A window is the primary input and output device of any UIMS application. It is an application's only means of access to the system display. A window is a combination of various features (such as title bar, menu bar, scroll bars and borders) that occupy a rectangle on the system display. You specify which of these features you want when you create a window. UIMS then draws and manages the window. Figure 2-2 shows the main features of a window.

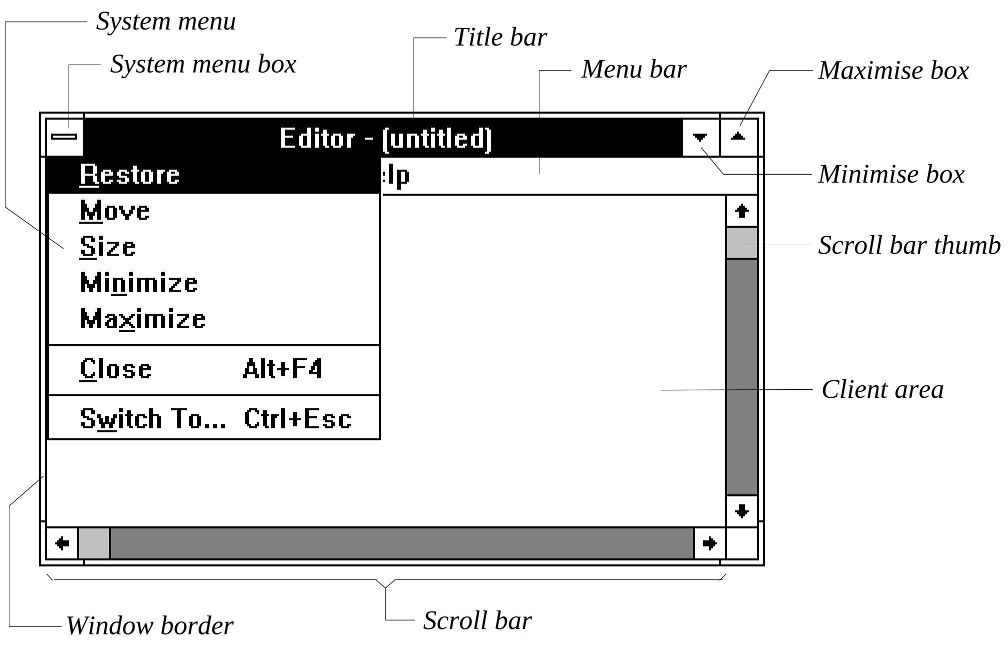


Figure 2-2. Window Features

Although an application creates a window and technically has exclusive rights to it, the management of the window is actually a collaborative effort between the application and UIMS. UIMS maintains the position and appearance of the window, manages standard window features such as the border, scroll bars and title, and carries out many tasks initiated by the user that directly affect the window. The application maintains everything else about the window. In particular, the application is responsible for managing the client area (the portion within the window borders). The application has complete control over the appearance of the window's client area.

To manage this collaboration, UIMS advises the application of changes that might affect the window. Because of this the application must have a message loop which separates out messages for different windows and carries out any processing that might be needed.

Menus

Menus are the principal means of user input in a UIMS application. A menu is a list of commands that the user can view and choose from. When you create an application, you supply the menu and command names. UIMS displays and manages the menus for you, and sends a message to your application when the user makes a choice. The message is the application's signal to carry out the command.

Dialog Boxes

A dialog box is a temporary window that you can display to allow the user to supply more information for a command. A dialog box contains one or more controls: that is, small windows that have very simple input or output functions – for example, an edit control is a simple window that lets the user enter and edit text. The controls in a dialog box let the user choose options, supply information and otherwise control the action of the command.

The Message Loop

Since your application receives input through a message queue, the main feature of any UIMS application is the "message loop". The message loop must retrieve messages from the application queue and process them appropriately.

Figure 2-3 shows how the UIMS and an application collaborate to process keyboard input messages. When the user presses and releases a key, this triggers a UIMS keypress event which processes the keyboard input. The event process constructs a keyboard input message which is placed in the application's message queue. The message loop must retrieve the message and route it to a subroutine that deals with all the messages for a particular window. This subroutine then uses the UIMS **DrawTextString** subroutine to display the character in the client area of the window.

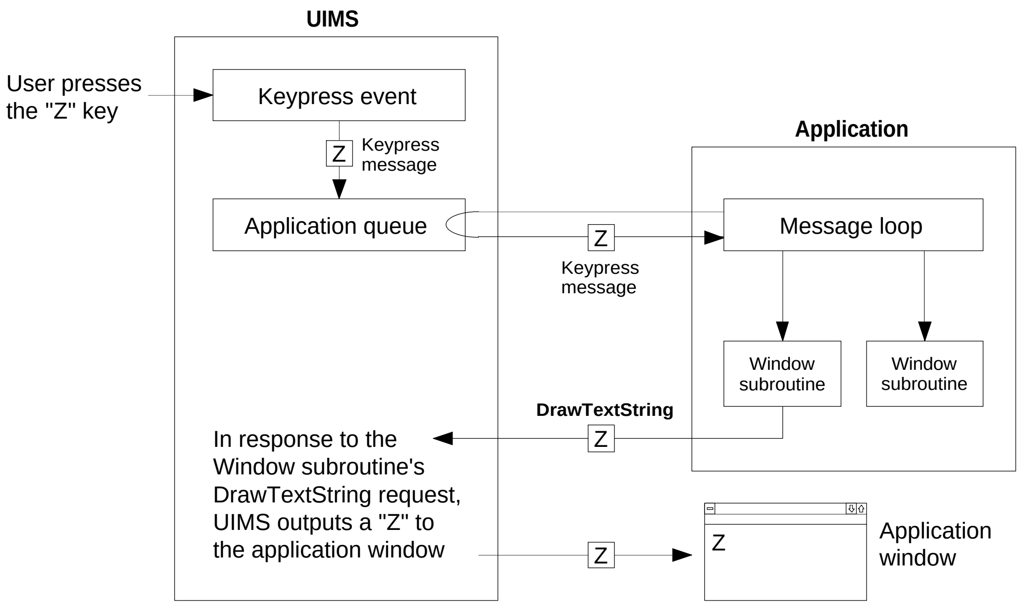


Figure 2-3. Processing Keyboard Input

Converting existing applications

NewView

Converting existing applications to UIMS can be a major job, involving as it does the provision of a message loop, windows, menu, dialog boxes, etc. However, RealLink provides a simple method of adding a graphical user interface to an existing character-based application.

NewView allows you to create menus and buttons, and set them up so that, instead of returning messages to the application, they return character strings. In a similar way, "hotspots" can be defined within a window, and set up to return character strings to the application. These two features allow the user to use a mouse in addition to the normal keyboard interface.

Chapter 3 describes how to use NewView to convert an existing character-based application.

Hybrid Applications

If you need to use UIMS features that are not available in NewView, but do not want to rewrite your application, you can create a Hybrid application. This will primarily be a character-based or NewView application, but will have some, generally self-contained, UIMS features.

Chapter 14 describes how to add UIMS modules to a NewView application.

Building a UIMS Application

To build a UIMS application, follow these steps:

1. Create DATA/BASIC source files that contain the message loop, window subroutines and other application code.
2. Create resource scripts that define the windows, menus, controls and other graphical elements required by your application.
3. Compile and catalog all DATA/BASIC sources on the host.
4. Use the Resource Compiler (RLRC) on the PC to compile the resource scripts.

Figure 2-4 shows the steps required to build a UIMS application.

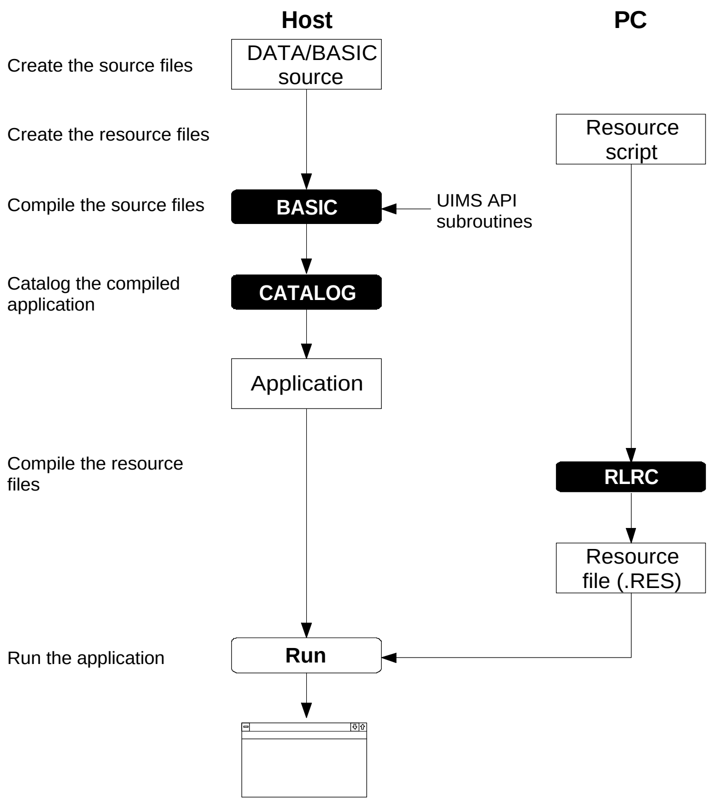


Figure 2-4. Building a UIMS Application

Chapter 3
Converting Existing Applications - NewView

This chapter explains how to use NewView to give an existing character-based application a graphical user interface, with only minimal changes to the source code. It describes a simple character-based application and then describes how to add various NewView features. The following points are covered:

- What is NewView?
- The components of a NewView application.
- Setting up NewView, and creating resources.
- Creating NewView groups
- Closing the Application
- Changing the options available to the user.
- Running Windows utilities from within your application.

What is NewView?

NewView is a subsystem of the RealLink User Interface Management System. It intercepts the UIMS messages which result from user actions and translates them into strings; these strings are then passed to the application as if they had been entered at the keyboard.

NewView allows the use of a pointing device, such as a mouse, in applications which previously could only be used from a keyboard. Operations which require the user to enter a predetermined character string can be made available on buttons, pull-down menus and hot-spots. A single click with the mouse can then initiate an operation which normally requires several keystrokes.

Contact Groups

Contacts are configured to return strings to the application by forming them into NewView contact groups. Note, however, that only **MenuItem** and **TitledButton** contacts can be used in groups, and that all the contacts in a group must be of the same type. Other types of contact that can be used in NewView applications are as follows:

AppWindow
ChildWindow
MenuBar
Menu
Text
Line
Rectangle

Hot-spots

The other major feature that NewView provides is the ability to define 'hot-spots' within the terminal window. These act in a similar way to button groups, generating text strings when clicked with the mouse. Hot-spots are defined by creating hot-spot groups.

The user can identify hot-spots by the shape of the mouse pointer; when pointing to a hot-spot, it changes to a hand shape.

The Components of a NewView Application

The following summarises the steps that must be added, when converting an application so that it can use NewView.

1. INCLUDE statements which specify the RealLink and UIMS constant definitions must be added.

2. The application must determine whether UIMS is available on the selected account and, if it is, initialise the UIMS system. This will also determine whether the application is running on RealLink or on a normal terminal (NewView can only be

used on RealLink). The remaining steps below must only be carried out if running on RealLink with UIMS available.

3. The application must sign on to UIMS.

4. Event masks must be set up to determine which types of message will be passed from UIMS to NewView (primary event mask), and which of these will be passed on to the application (secondary event mask). NewView applications will not normally include a message loop, and the secondary event mask should therefore disable all types of message.

5. If synchronous error handling is required, this must be selected.

6. Graphics coordinate mode must be selected.

7. The UIMS resources (windows, buttons, menu items and other contacts) used by the application must be created. To minimise changes to the application, this could be done by a separate cataloged subroutine, or the resources could be loaded from a compiled resource script on the PC.

8. If a window other than the RealLink window is to be used as the terminal window, the terminal window functionality must be transferred to the required window.

9. NewView contact and hot-spot groups must be created.

10. Each time the application displays a different screen, the appropriate NewView groups should be enabled and disabled as appropriate.

11. When the application terminates, if it is running on RealLink, the following must be done:

    - All NewView contact and hot-spot groups must be destroyed.

    - If a window other than the RealLink window has been used as the terminal window, the terminal window must be returned to RealLink.

    - The application must be signed off from UIMS.

The MENUEX Character-based Application

In order to demonstrate how to convert an application using NewView, we need an application to convert. The one we have chosen simply presents a series of menus to the user, and would normally form part of a much larger application. However, it illustrates well the way in which NewView can improve the user interface and make an application simpler to use.

The character-based application is called MENUEX; its DATA/BASIC source code is listed in Appendix A. It works by reading menu definitions from a file. These definitions consist of lists of item numbers, screen column positions, screen row positions, item titles and links to sub-menus. In addition, there is a title for the complete menu. In use, the user enters the number of the item required; or E to return to the previous menu, M to return directly to the main menu, or OFF to return to TCL.

The menu definition file contains only six menus. There is a main menu, the first four items of which are linked to four sub-menus. The first item on sub-menu 1 is linked to a fifth sub-menu, and the first four items on sub-menu 4 all return the user to the main menu. In a working application, there would be more menus, and the sub-menu links would, where appropriate, be replaced with identifiers for data-entry screens and processing routines.

The main menu displayed by this application is shown in Figure 3-1.

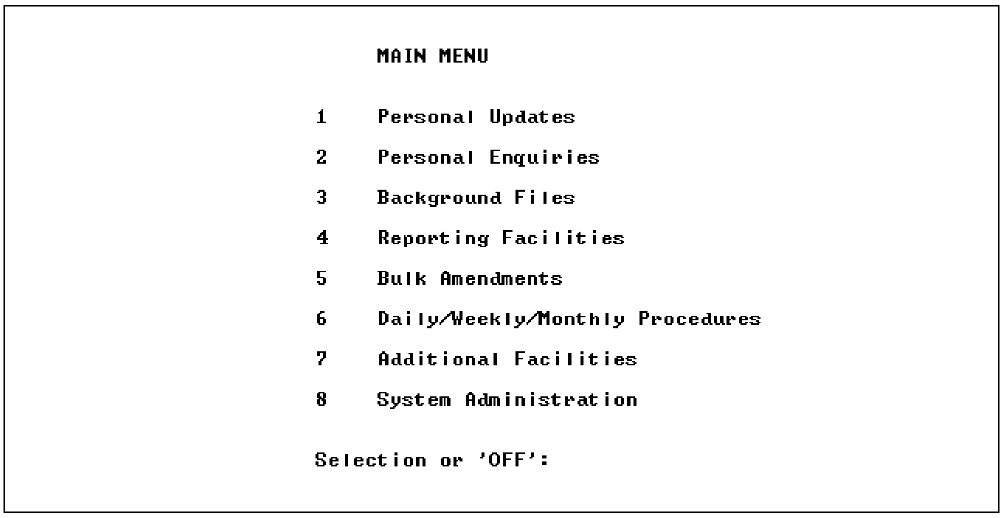

```
          MAIN MENU

     1    Personal Updates

     2    Personal Enquiries

     3    Background Files

     4    Reporting Facilities

     5    Bulk Amendments

     6    Daily/Weekly/Monthly Procedures

     7    Additional Facilities

     8    System Administration

     Selection or 'OFF':
```

Figure 3-1. Example Character-based Application

The MENUEX Source Code

The code for MENUEX is listed in Appendix A. It is divided into three routines:

- A main routine which initialises the application, displays the selected menu and processes the user selection.
- The BUILD subroutine which reads a menu definition from the menu definition file and converts this into a string which can be displayed on the screen.
- The ERRSUB subroutine which displays error messages when necessary.

The Main Routine

The main routine is divided into four sections:

- An introductory section which initialises the application and opens the menu definition file.
- An outer loop which calls the BUILD subroutine to set up the selected menu and then displays the result. As each menu is selected, its identifier is added to a 'history' array (PREVID), which is used when returning to the previous level.
- An inner loop which waits for the user to make a selection and takes the appropriate action.
- A closing section which clears the screen and returns the user to TCL.

The inner loop contains a case structure which processes the user's selection as follows:

- If a valid menu item is selected, the ID variable points to the selected menu for display on the next pass through the loop.
- If the user enters M, the ID variable is set to the main menu for display on the next pass through the loop. The history variable (PREVID) is cleared.
- If the user enters E, the identifier of the previous menu is retrieved from PREVID.
- If the user enters OFF, the control variable (OK) for the outer loop is cleared, so that the program will drop out into the closing section.
- If the user has entered none of the above, the ERRSUB subroutine is called to display an error message. The variable containing the user selection (ANS) is then cleared, so that the program will remain within the inner loop until a valid response is entered.

The BUILD Subroutine

The menu for display is built up in the variable SCR. The BUILD routine starts by clearing this variable and then reads the details of the selected menu into a dynamic array (MENUREC). If there is no definition for the selected menu, an error message is displayed and the subroutine returns.

The new menu starts with the code to clear the screen, followed by the position and text of the menu heading. The routine then loops, reading the details of each menu item in turn from MENUREC, and adding them to SCR in the appropriate form: a cursor positioning code, followed by the item number; then another cursor positioning code, followed finally by the text of the item. Two variables, START.OPT.NO and END.OPT.NO, store the range of valid item numbers for testing in the main routine.

If the item identifier is 'S', the item is assumed to be the prompt which appears at the bottom of the menu. This is added to SCR as a cursor positioning code and the text of the item, and two variables, SELCOL and SELROW, are set to specify the screen position for the INPUT statement in the main routine.

The ERRSUB Subroutine

This subroutine simply prints an error message (from the ERRMSG variable) in reverse video. It then waits for the user to press a key before clearing the message and returning to the calling routine.

The NewView Version – MENUNV

The NewView version of MENUEX is called MENUNV (see Appendix A for its source code listing). The differences between the two applications fall into two categories:

- NewView enhancements to existing facilities.
- Additional facilities added to illustrate NewView features.

The enhancements to existing facilities are as follows:

- The definition of a hot-spot for each item on the displayed menu. These allow the user to select an item simply by pointing to it with the mouse and clicking the left mouse button.
- The addition of Back and Main buttons at the top of the application's window to provide easy access to the E and M commands.
- The addition of a menu bar at the top of the window. This has a File pull-down menu containing an Exit command which has the same effect as the OFF menu command.

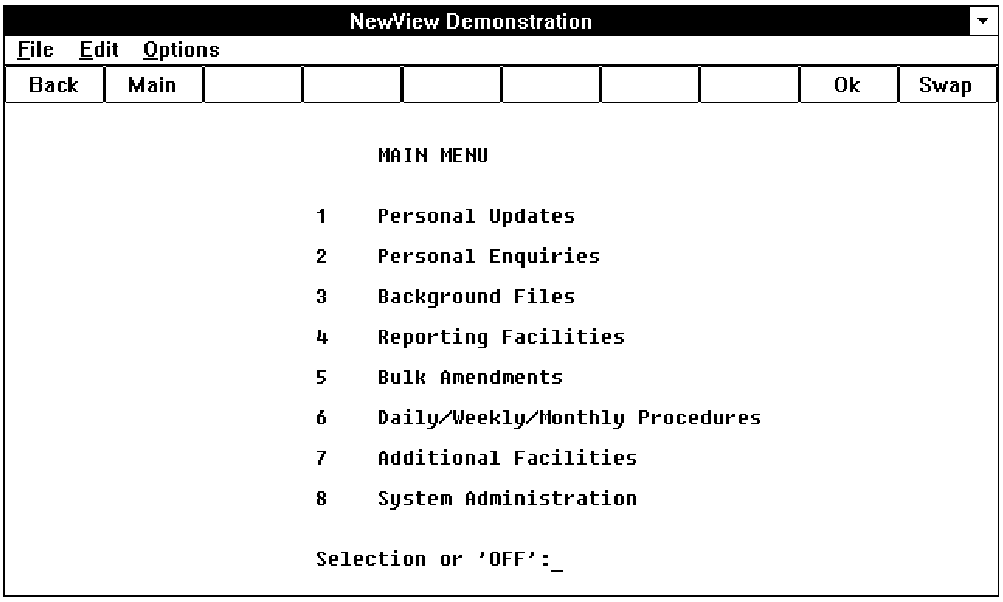
Note that these features do not replace those provided by MENUEX. The application can still be used from the keyboard as described in the previous section. Also, the NewView code is added in such a way that MENUNV can be used on a normal terminal.

The additional facilities are as follows:

- The provision of the RealLink Print commands on the File pull-down menu.
- The addition of an Edit pull-down menu containing the RealLink Edit commands.
- The addition of a third, Options pull-down menu containing additional commands.
- The addition of eight extra buttons which could be used for more commands.
- Example routines which show how the NewView features can be changed from screen to screen, or in response to user input.

Note: In order to distinguish between the character-based menus displayed by MENUNV and the NewView menus on the menu bar, the latter will be referred to as pull-down menus.

Figure 3-2 shows the main menu for the MENUNV application.



```
                    MAIN MENU

               1    Personal Updates
               2    Personal Enquiries
               3    Background Files
               4    Reporting Facilities
               5    Bulk Amendments
               6    Daily/Weekly/Monthly Procedures
               7    Additional Facilities
               8    System Administration

               Selection or 'OFF':_
```

Figure 3-2. Example NewView Application

The sections which follow describe in detail the changes which must be made to MENUEX to enable it to use NewView. The source code for the converted application (called MENUNV) is listed in Appendix A.

Constant Definitions

The appropriate UIMS and NewView constant definitions must be included at the beginning of the application. These are contained in three header items – RFWDEFS, UIMSDEFS and UIMSCOMMON – in the file UIMS-TOOLS. Note that all NewView applications need RFWDEFS, but if only hot-spots are used, the other two can be omitted.

The constant definitions for MENUNV are as follows:

```
INCLUDE RFWDEFS FROM UIMS-TOOLS
INCLUDE UIMSDEFS FROM UIMS-TOOLS
INCLUDE UIMSCOMMON FROM UIMS-TOOLS
```

In addition, a header file, MENUNV.H, is included. This contains definitions specific to MENUNV, many of which are common to both the DATA/BASIC source and the resource script (see page 3-11).

```
INCLUDE MENUNV.H
```

Initialising UIMS

NewView can only be used if the terminal supports UIMS (that is, if the application is being run on RealLink for Windows), and if UIMS is available in the selected account. The **InitialiseUims** subroutine sets a common variable, UIMS.CAPABLE, which can be tested to determine whether the terminal supports UIMS. However, in order to run **InitialiseUims**, you must have access to this subroutine. The following code tests for the availability of UIMS and, if it is, runs **InitialiseUims**:

```
UIMS.CAPABLE = FALSE   ;* until we've tried, assume no UIMS support.
* See if the InitialiseUims subroutine exists. If it does, call it.
OPEN "MD" TO ACCMD THEN
   READ ACCREC FROM ACCMD,"InitialiseUims" THEN CALL InitialiseUims
   CLOSE ACCMD
END
```

Note: If you are only using hot-spots, there is no need to initialise UIMS. Instead, you should call the **IsUimsCapable** subroutine (checking that it exists, as shown above). Refer to the *UIMS DATA/BASIC API, Reference Manual* for details.

Signing On to UIMS

Once you have initialised UIMS for your application, you must sign on in order to create a application context. The context handle returned when you sign on must be passed to many of the UIMS subroutines and will also be needed when you sign off.

The **SignOn** subroutine requires two parameters: a string containing the name of the application and the name of a variable in which to return the handle of the application context. For example:

```
CALL SignOn("MENUNV", CONTEXT)
```

You should then test the context handle to check that the application is properly signed on. A context of zero indicates an error.

```
IF NOT(CONTEXT) THEN
   UIMS.CAPABLE = FALSE ;* can't sign on, so can't use NewView
   ERRMSG = "Error - failed to signon to UIMS"
   GOSUB ERRSUB
   RETURN
END
```

ERRSUB is a routine that will display an error message. Note that the UIMS.CAPABLE variable is set to FALSE to indicate that NewView cannot be used; the application can, however, be run on a normal terminal and can therefore continue without NewView.

Note: If you are using only hot-spots, there is no need to sign on to UIMS.

Event Masks

One of the reasons it is easy to convert applications using NewView is that no message loop is needed. NewView itself processes most of the messages generated by UIMS and these are never passed on to the application. However, in order for NewView to work correctly, two event masks must be set up to ensure that NewView receives the correct types of message, and that unwanted messages are disabled. This is done as follows:

1. A primary event mask is set up using the **SetEventMask** subroutine. This ensures that NewView receives the correct types of message.

2. A secondary event mask is set up using the **SetSecondaryEventMask** subroutine. This ensures that any messages passed on by NewView to the RealLink terminal emulator do not reach the application.

The following code sets these event masks:

```
* set an event mask that will allow NewView the right messages to
* process
CALL SetEventMask(CONTEXT❶, CONTEXT❷, UIMS.EM.NEWVIEW❸, ERR❹)
* disable unwanted messages
CALL SetSecondaryEventMask(CONTEXT❶, 0❺, FALSE❻, FALSE❼, ERR❹)
```

❶ In both subroutines, the first parameter is the handle of the application context, returned by the **SignOn** subroutine.

❷ A primary event mask can be set for individual contacts as well as for a complete application. The second parameter to **SetEventMask** specifies this contact. In this case, we require a mask for the complete application, so this parameter is the handle of the application context.

❸ This parameter is the NewView event mask, defined in the UIMSDEFS header.

❹ The final parameter to both subroutines must be a variable in which a completion code can be returned. If required, this variable can be tested to see if an error has occurred.

❺ This parameter specifies those types of message which should be allowed to reach the application. A value of zero disables all messages.

❻ This parameter allows you to disable messages which cannot be masked using **SetEventMask**. A value of FALSE disables all non-maskable messages.

❼ The fourth parameter is provided for use in future versions of UIMS. It must be included, but its value will be ignored.

Error Handling

Once you have enabled or disabled UIMS error messages, you must select the error handling mode. Most UIMS and NewView subroutines have a *vErr* parameter in which a completion code is returned.

- If synchronous error handling is selected, this parameter will return set to UIMS.SUCCESS (zero) for successful completion, or one of the error codes defined in UIMSDEFS and RFWDEFS if an error has occurred.
- If asynchronous error handling is selected, *vErr* always returns UIMS.SUCCESS and errors are reported by means of a UIMS message. Since NewView applications do not have a message loop, you cannot use asynchronous error handling.

The error handling mode is set as follows:

```
CALL SetSync(CONTEXT, TRUE, ERR)
```

The second parameter controls the error mode; TRUE selects synchronous error handling.

Coordinate Mode

NewView applications must use graphics coordinate mode. This is set as follows:

```
CALL SetCoordMode(CONTEXT, UIMS.COORD.GRAPHIC, ERR)
```

Creating Resources

The resources for a NewView application consist of all the windows, pull-down menus and buttons presented to the user. These can be created in two ways:

- They can be pre-defined by preparing a resource script and compiling it using the UIMS Resource Compiler.
- They can be created dynamically during initialisation or as required by the application.

For MENUNV we will use a resource script. Note, however, that whichever method you use, you can modify your resources dynamically at any time, as required by your application.

The resources required by MENUNV consist of a main application window (AppWindow), a Child window to use as the terminal window, a menu bar with three pull-down menus, and ten titled buttons.

Creating a Resource Script

Resources are defined on the PC in a source file (resource script) which can be produced by any ASCII text editor (Windows Notepad, for example). The completed source must then be compiled, using the UIMS Resource Compiler, RLRC.

The following shows the main elements of the resource script for MENUNV. The complete resource script is shown in Appendix A.

```
#INCLUDE RFWDEFS.H ❶
#INCLUDE MENUNV.H ❶

* Main application window and pull-down menus
APPWINDOW = APPWIN ❷
{
   TITLE = 'NewView Demonstration'
   POSITION = 0,40 ❸
   SIZE = 1000, 825 ❸
   STYLE = NOSCROLL, MOVABLE, ICONISABLE ❹
   BDRSTYLE = BORDER ❺

   MENUBAR = APPMENUBAR ❻
   {
      * The File pull-down menu has all the RealLink Printing
      * commands plus Exit
      MENU = APPMENUFILE ❼
      {
         TITLE = '&File'
         ENABLED = TRUE
         CHILDREN = '&Print'          = ID.FILEPRINT,
                    'Print &Window'   = ID.FILEPRINTWINDOW,
                    'P&rinter Setup' = ID.FILEPRINTERSETUP,
                    '-'               = 0,
                    'E&xit'           = APPMENUEXIT ❽
      }

      .
      . * definitions for Edit and Options pull-down menus
      .

   * Child window to use as the terminal window
   CHILDWINDOW = CHILDWIN ❾
   {
      POSITION = 0, TEWINSTART
      SIZE = 1000, 755
      STYLE = NOSCROLL
      BDRSTYLE = NONE
      EVENTMASK = NEWVIEW
   }
   * Buttons

   TITLEDBUTTON = BUT1 ❿
   {
      TITLE = 'Back'
      POSITION = BUT1X, BUTY
      MAPPED = FALSE
      ENABLED = TRUE
      SIZE = BUTWIDTH, BUTHEIGHT
   }

   .
   . * definitions for nine more buttons
   .
}
```

❶ These two lines insert the contents of header files containing the definitions of constants used in the application. RFWDEFS contains definitions used by NewView, and must be copied onto the PC from the host and given the extension '.H' (RFWDEFS can be found in the host file UIMS-TOOLS). MENUNV.H contains definitions specific to this application which are used in the resource script and the DATA/BASIC program (see Appendix A).

❷ This line introduces the definition for the application's main App window. The window is given the identifier APPWIN, which has been assigned a numeric value in the header file MENUNV.H.

❸ The next two lines specify the window's initial position on the screen and its size. These values will be interpreted as character or pixel units, depending on the current coordinate mode of the application context. If graphics mode is selected, the coordinates are interpreted as pixel positions on an arbitrary screen 1000 units wide by 1000 units high. When the resources are loaded into an application, the coordinates are scaled to fit within the actual screen.

MENUNV will run in graphics mode, so the position is at the left-hand edge and 40 units down, and the window size is 1000 units wide (full width) by 825 high.

❹ This line specifies the style of the window. It consists of a combination of a number of style elements. In this case the elements MOVABLE, NOSCROLL and ICONISABLE are selected, producing a window without scroll bars that can be moved and minimised to an icon.

❺ The next line specifies whether or not the window has a border. It can have the value BORDER as shown above, or NONE to specify no border. Note however, that the window style controls the type of border and in some cases will automatically generate a border, even though none is specified. Refer to the *UIMS DATA/BASIC API, Reference Manual* for more details. In this case, the setting of the border style parameter is irrelevant because the MOVABLE style element automatically generates a single border. The parameter must be included, however, and setting it to BORDER will serve as a reminder that the window has a border.

❻ This line introduces the definition for the App window's menu bar. Because the definition is nested within that of the App window, the menu bar will be automatically attached to the App window.

❼ This line introduces the definition for the File pull-down menu. The nested definition makes the menu a child of the menu bar. Note that the title includes an ampersand (&) preceding the letter 'F'; this specifies that this letter may be used in combination with the ALT key to select the pull-down menu from the keyboard.

❽ This line defines the items that will appear on the File pull-down menu. Note that the three Print items are given identifiers (defined in RFWDEFS.H) which allow them to call the standard RealLink printing functions. The Exit item, however, is handled by the application and has an identifier defined in MENUNV.H.

The item defined as `'-'=0` creates a line separating Exit from the three Print items.

The definitions for the other two pull-down menus are similar (see the complete resource script in Appendix A).

❾ This line introduces the definition for the Child window. This has no scroll bars or border, and is the same width as the App window's client area. Its height, however, is smaller than that of the App window, and it is positioned 50 units (TEWINSTART) down from the top of the App window to leave room for the titled buttons. Note that because this window will be used as the terminal window, it must be given the same NEWVIEW event mask as the application.

❿ This line introduces the definition for the first of ten titled buttons that will form a 'button bar' just below the menu bar. The definitions for the other nine buttons are similar (see the complete resource script in Appendix A).

Loading the Compiled Resources

The compiled resources are loaded into a host UIMS application by calling the **LoadAppRes** subroutine. The following line loads the resources for the MENUNV application:

```
CALL LoadAppRes(CONTEXT❶, "menunv.res"❷, ERR❸)
```

❶ This parameter is the handle of the application context.

❷ This is the name of the compiled resource file on the PC. If no path is specified, it is assumed to be in the resource directory specified in the RFW.INI file.

❸ This is a variable in which to return the completion status of the subroutine. You should test this variable to check that the resources have been created successfully. A non-zero value indicates an error.

```
IF ERR THEN
   UIMS.CAPABLE = FALSE
   CALL SignOff(CONTEXT, ERR)
   ERRMSG = "Cannot find MENUNV.RES resource file"
   GOSUB ERRSUB
   RETURN
END
```

Note that without resources there is no point is remaining signed on to UIMS. The UIMS.CAPABLE variable is set to FALSE to indicate that NewView cannot be used. The application can, however, be run on a normal terminal and can therefore continue without NewView.

Displaying the Application Window

When the application window is complete, it can be displayed. This is done by adding the window as a child of the application context by calling the **AddChild** subroutine.

AddChild requires the following parameters:

- The handle of the application context.
- The handle of the parent object; that is the object to which the child is to be attached.
- A number indicating where, in any existing list of children, the new child is to be added. This allows, for instance, a new menu item to be added at the beginning of a pull-down menu rather than at the end.
- The handle of the child to be added.

- A variable in which to return any error code.

The following example attaches the application window to the context of the MENUNV application.

```
CALL AddChild(CONTEXT, CONTEXT, -1❶, APPWIN, ERR)
```

❶ The list index of -1 adds the new child at the end of any existing list.

Setting the Terminal Window

Once you have displayed the App window, you can transfer the terminal functionality to the appropriate window – in this case the Child window created in the resource script. This will also hide the RealLink window.

To hide the RealLink window, include the following line in your application:

```
CALL SetTeWindow(CONTEXT, CHILDWIN❶, UIMS.NONE❷, ERR)
```

❶ This is the identifier for the window which will become the terminal window.

❷ This parameter specifies that the RealLink window is to be hidden.

Note: Before returning to TCL, your application must return terminal functionality to the RealLink window (see page 3-21).

You must then ensure that keyboard input is directed to the new terminal window by giving it the focus:

```
CALL SetContactFocus(CONTEXT, CHILDWIN, ERR)
```

Creating NewView Groups

Before they can be used in a NewView application, the UIMS contacts you have created must be formed into groups. The main reason for doing this is to assign text strings to the contacts; each contact will then return its associated text string to the application whenever it is operated.

Three types of NewView group are required in the MENUNV application: a menu item contact group for each pull-down menu on the menu bar; contact groups for the titled buttons on the button bar; and a hot-spot group for each different menu displayed.

Contact Groups

The contact groups – two menu item groups (see note 1 below), and two button groups – are all created in the same way: by calling the **CreateNVContactGroup** subroutine. The following example shows how the group for the Options pull-down menu is created.

```
EQU MENU2GRPID     TO 5 ❶
EQU APPMENUDIARY   TO 150 ❷
EQU APPMENUCALC    TO 151
EQU APPMENUCHANGE  TO 152
MENU2GRP.RESP=DIARY.RESP:CRET:AM:CALC.RESP:CRET:AM:SWAP.RESP:CRET ❹

CALL CreateNVContactGroup(CONTEXT, ...
                          MENU2GRPID❶, ...
                          APPMENUDIARY❷, ...
                          3❸, ...
                          MENU2GRP.RESP❹, ...
                          ERR)
```

❶ This parameter is a unique number identifying the group.

❷ This is the numeric identifier for the first contact in the group. Note that the contacts must be consecutively numbered.

❸ This is the number of contacts in the group

❹ This is a dynamic array, where each attribute contains the text string to be returned by the corresponding contact in the group. Note that each string ends with a carriage return, as if it had been entered at the keyboard.

Notes:

1. In MENUNV, only the File and Options pull-down menus require NewView groups; the Edit pull-down menu is managed entirely by RealLink. Similarly, the File pull-down menu group consists of only one item, Exit, since the Print items are all managed by RealLink. Separator items should not be included in groups.

2. The MENUNV button bar is divided into two NewView groups. The first consists of the two leftmost buttons: Back and Menu. The second contains the remaining eight buttons, though only the two rightmost, OK and Swap, are used. This division allows the application to disable the OK and Swap buttons when the Main menu is displayed.

Hot-spot Groups

NewView hot-spots are areas defined in the terminal window, with which text strings are associated. When the user points to a hot-spot with the mouse and clicks the left-hand button, the associated text string is returned to the application as if it had been entered at the keyboard. For example, suppose the user must enter the letter 'Q' to select a particular item from a menu. A NewView hot-spot could be set up surrounding the text of that item as it is displayed in the terminal window, with the string `'Q':CHAR(13)` assigned to it. Then, all the user would need to do to select that item, would be to point to the item with the mouse and click; the assigned string would be sent to the application, selecting the item in the same way as if 'Q' had been entered at the keyboard.

Hot-spots are created with the **CreateNVHotspotGroup** subroutine. This requires the following parameters:

- The handle of the application context.
- A unique numeric identifier for the group to be created.
- The number of hot-spots in the group
- Five dynamic arrays containing respectively the horizontal positions, vertical positions, widths, heights and text responses for the hot-spots in the group. The positions and sizes of the hot-spots must be defined in *text* coordinates.
- A variable in which to return a completion code.

The details of the hot-spots required are therefore built up in five dynamic arrays, while a sixth variable counts the hot-spots as they are set up. The setting up of the hot-spots is in two stages:

- First any previous hot-spots are destroyed, the dynamic arrays are cleared, and the hot-spot count is reset to zero.

```
IF UIMS.CAPABLE THEN
   * destroy any previous hot-spot group
   IF HOTSPOTS # 0 THEN ...
       CALL DestroyNVGroup(CONTEXT, HOTGRPID, ERR)

   * reset hot-spot attribute arrays
   XPOS = '' ; YPOS = ''      ;* positions
   WIDTH = '' ; HEIGHT = ''   ;* sizes
   RESP = ''                  ;* responses
   HOTSPOTS = 0              ;* no hot-spots now exist
END
```

- Then, for each new hot-spot, the appropriate attributes are appended to the arrays, and the hot-spot counter is incremented.

```
IF UIMS.CAPABLE THEN
   XPOS<-1> = COL ; YPOS<-1> = ROW             ;* position
   WIDTH<-1> = LEN(TITLE)+5 ; HEIGHT<-1> = 1   ;* size
   RESP<-1> = OPT.NO:CRET                      ;* response

   HOTSPOTS = HOTSPOTS+1   ;* increment the hot-spot counter
END
```

  The menu item position, text and response are read from the Menu Definition File when the character-based menu is set up. Note that these can be used directly, since, although MENUNV is operating in graphics coordinate mode, hot-spots must always be defined using *text* coordinates.

Once all the hot-spots have been set up, **CreateNVHotspotGroup** can be called.

```
CALL CreateNVHotspotGroup(CONTEXT, ...
                          HOTGRPID❶, ...
                          HOTSPOTS, ...
                          XPOS, ...
                          YPOS, ...
                          WIDTH, ...
                          HEIGHT, ...
                          RESP, ...
                          ERR)
```

❶ This is the numeric identifier for the hot-spot group created.

In MENUNV, each menu displayed requires different hot-spots. A hot-spot group is therefore set up when the menu is constructed (in the BUILD subroutine), and created when the menu is displayed.

Error Messages

A feature of UIMS that can be used in NewView applications is the use of message boxes to display error messages. A message box is created with the **CreateMessageBox** subroutine. This requires the following parameters:

- The handle of the application context.

- The type of message box required.
- The title of the message box.
- The message to be displayed.
- A list of button names (to replace the default names if required).
- A variable that will be used to return the selected button.
- A variable in which to return the completion status of the subroutine.

In MENUNV, the ERRSUB subroutine is modified to use a message box if UIMS is available:

```
ERRSUB:
   IF UIMS.CAPABLE THEN
      CALL CreateMessageBox(CONTEXT❶, ...
                            UIMS.INFO❷, ...
                            "Error"❸, ...
                            ERRMSG❹, ...
                            ""❺, ...
                            REPLY❻, ...
                            ERR❼)
      * Set the focus back to the terminal window
      CALL SetContactFocus(CONTEXT, CHILDWIN, ERR) ❽
   END ELSE
      .
      . ;* if UIMS not available display the message below the menu
      .
   END
RETURN
```

❶ The first parameter is the handle of the application context, returned by the **SignOn** call.

❷ The second parameter specifies the message box type. **UIMS.INFO** specifies a box with an information icon and a single button. The button is labelled OK unless changed by providing a title in parameter ❺.

❸ The next parameter specifies the title of the message box.

❹ The next parameter specifies the message displayed in the box. In MENUNV this is set up by the calling routine.

❺ The next parameter is a list of titles to use for the buttons. In this case, the parameter is a null string and the default title (OK) will therefore be used.

❻ The penultimate parameter is a variable in which the selected button in returned. The about box has only one button and the value returned will therefore, in this case, be ignored.

❼ The final parameter must be a variable in which any error code will be returned. A return value of 0 indicates successful completion.

❽ When the message box has been removed, the focus will normally be passed to the main App window. This line ensures that control is returned to the application's terminal window.

Closing the Application

Before leaving a NewView application, there are three tasks that must be carried out:

- You must destroy all contact and hot-spot groups.
- You must transfer the terminal window back to RealLink.
- You must sign off from UIMS.

The following shows how this is done in the MENUNV application.

```
IF HOTSPOTS # 0 THEN CALL DestroyNVGroup(CONTEXT, HOTGRPID, ERR)  ❶

CALL DestroyNVGroup(CONTEXT, BUT1GRPID, ERR)
CALL DestroyNVGroup(CONTEXT, BUT2GRPID, ERR)

CALL DestroyNVGroup(CONTEXT, MENU1GRPID, ERR)
CALL DestroyNVGroup(CONTEXT, MENU2GRPID, ERR)

CALL SetTeWindow(0, 0, TE.SHOWWIN, ERR)  ❷

CALL SignOff(CONTEXT, ERR)  ❸
```

❶ This line destroys the hot-spot group, if one exists. The **DestroyNVGroup** subroutine requires the handle of the application context, the identifier of the group to be destroyed, and a variable in which to return a completion code. The contact groups are destroyed in the same way.

❷ This line returns the terminal window to RealLink. The zero values of the first two parameters specify the RealLink context and window respectively. The third parameter specifies that the RealLink window should be displayed.

❸ This line signs off the application from UIMS. The first parameter must be the context handle returned by the **SignOn** call.

Changing the Options Available

There are many reasons why you might want to change the options available within your NewView application. The most obvious one has already been addressed above; that is, to allow for running the application on both RealLink and a normal terminal. Two other possibilities are: offering different options according to the menu displayed; and giving the application different modes of operation and allowing the user to choose between them. Examples of both these possibilities are described below.

Enabling and Disabling Resources

When the MENUNV resources are created, the Options pull-down menu and the OK and Swap buttons are all disabled. Selecting a different menu enables these facilities, while returning to the main menu disables them again. This is done by adding some additional code to the outer loop of the main routine, as follows:

```
   .
   .
   .
   REPEAT               ;* end of inner loop

   IF UIMS.CAPABLE THEN GOSUB CHANGE.MENUBAR ❶

REPEAT               ;* end of outer loop
   .
   .
   .
CHANGE.MENUBAR:
   * if the main menu is displayed
   IF ID = "MAIN" THEN
      * disable button group 2
      CALL SetEnabledNVGroup(CONTEXT, BUT2GRPID, FALSE, ERR) ❷
      * disable the Options pull-down menu
      CALL SetEnabledNVGroup(CONTEXT, MENU2GRPID, FALSE, ERR) ❸
      CALL SetEnabled(CONTEXT, APPMENUOPTIONS, FALSE, ERR) ❹

   END ELSE
      * otherwise enable button group 2
      CALL SetEnabledNVGroup(CONTEXT, BUT2GRPID, TRUE, ERR) ❺

      * and enable the Options pull-down menu
      CALL SetEnabledNVGroup(CONTEXT, MENU2GRPID, TRUE, ERR) ❺
      CALL SetEnabled(CONTEXT, APPMENUOPTIONS, TRUE, ERR) ❺
   END
RETURN
```

❶ The pull-down menu and button resources will only exist if UIMS is available, so the CHANGE.MENUBAR subroutine is only called if UIMS.CAPABLE is set.

❷ This line disables the button group when the main menu is displayed. The second parameter is the identifier of the button group, and the third the required enabled or disabled state; FALSE disables the group.

❸ This line disables the group of menu items on the Options pull-down menu.

❹ Disabling the menu items would give us a pull-down menu with all its commands disabled. This line disables the pull-down menu as well. Note that, as the pull-down menu is not part of a group, we use the **SetEnabled** subroutine instead of **SetEnabledNVGroup**.

❺ These lines enable the buttons and the Options pull-down menu when any other menu is displayed.

Changing Existing Resources

When the MENUNV resources are created, the leftmost button on the button bar is titled 'Back', and the one next to it, 'Main'. The second button group contains a button titled 'Swap', which swaps these two buttons, so that the leftmost becomes the Main button and that next to it, the Back button.

Note: The Options pull-down menu also contains a Swap command. This performs the same function as the Swap button.

The swap function is made available by adding an additional condition to the CASE statement in the inner loop, as follows:

```
    .
    .
    .
BUT1GRP.RESP = BACK.RESP:CRET:AM:MAIN.RESP:CRET ❶
BUT1GRP.RESP2 = MAIN.RESP:CRET:AM:BACK.RESP:CRET ❶
    .
    .
    .
CASE ANS = SWAP.RESP ❷
   IF UIMS.CAPABLE THEN
      GOSUB SWAP.BUTTONS ❸
   END ELSE
      ERRMSG = "Invalid entry : ":ANS
      GOSUB ERRSUB
   END
   ANS = 0 ❹
    .
    .
    .
SWAP.BUTTONS:
   IF SWITCH = 0 THEN ❺
      CALL TitledButtonSetTitle(CONTEXT, BUT1, "Main", ERR) ❻
      CALL TitledButtonSetTitle(CONTEXT, BUT2, "Back", ERR) ❻

      CALL ChangeNVContacts(CONTEXT, BUT1GRPID, BUT1, 2,
BUT1GRP.RESP2, ERR) ❼

      SWITCH = 1 ❺

   END ELSE
      CALL TitledButtonSetTitle(CONTEXT, BUT2, "Main", ERR) ❻
      CALL TitledButtonSetTitle(CONTEXT, BUT1, "Back", ERR) ❻

      CALL ChangeNVContacts(CONTEXT, BUT1GRPID, BUT1, 2,
BUT1GRP.RESP, ERR) ❼

      SWITCH = 0 ❺
   END
RETURN
```

```
.
.
.
```

❶ These two lines define alternative responses for button group 1.

❷ This is the response assigned to the Swap button and the Swap pull-down menu item.

❸ The buttons will only exist if UIMS is available, so the SWAP.BUTTONS subroutine is only called if UIMS.CAPABLE is set. Otherwise, the selection is rejected as invalid.

❹ The control variable for the inner loop is reset, so that we continue with the same menu.

❺ The SWITCH variable is toggled each time the buttons are swapped, so that next time the command is used, the program knows the current state.

❻ These lines change the titles of the buttons. In each case, the second parameter is the identifier for the button concerned, and the third, the new title.

❼ These lines change the responses for the buttons in the first group. The **ChangeNVContacts** subroutine requires the following parameters:

- The handle of the application context.
- The identifier for the required contact group.
- The handle of the first contact in the group to be changed.
- The number of contacts to be changed.
- A dynamic array containing the new responses for the contacts to be changed.
- A variable in which a completion code can be returned.

Running Utilities

In some cases, applications give the user access to useful utilities or sub-programs. This can also be done with NewView. MENUNV includes a Calculator command on the Options pull-down menu, which runs the Windows Calculator program. This is done by adding an additional condition to the CASE statement in the inner loop, as follows:

```
.
.
.
CASE ANS = CALC.RESP ❶
   IF UIMS.CAPABLE THEN
      GOSUB RUN.CALC ❷
   END ELSE
      ERRMSG = "Invalid entry : ":ANS
      GOSUB ERRSUB
   END
   ANS = 0 ❸
        .
        .
        .
RUN.CALC:
   COMMANDLINE = "calc.exe" ❹
   WINDOWSTATE = EXECUTE.SHOWNORMAL ❺
   CONTROL = EXECUTE.WAIT ❻
   CALL Execute(COMMANDLINE❹, WINDOWSTATE❺, CONTROL❻, ERR)
   IF ERR THEN ❼
      ERRMSG = "Unable to run Calculator : ":ERR
      GOSUB ERRSUB
   END
   CALL SetContactFocus(CONTEXT, CHILDWIN, ERR) ❽
RETURN
```

❶ This is the response assigned to the Calculator item on the Options pull-down menu.

❷ The pull-down menu will only exist if UIMS is available, so the RUN.CALC subroutine is only called if UIMS.CAPABLE is set. Otherwise, the selection is rejected as invalid.

❸ The control variable for the inner loop is reset, so that we continue with the same menu.

❹ The **Execute** subroutine is used to run Windows programs on the PC. This parameter is the command line which runs the Windows Calculator.

❺ This parameter determines the appearance of the Calculator window. **EXECUTE.SHOWNORMAL** produces a normal, active window.

❻ This parameter specifies that control should not return to MENUNV until the Calculator is closed.

❼ If, for any reason, it is not possible to run the Calculator, the ERR variable is returned containing an error code. If this occurs an error message is displayed.

❽ When the Calculator is closed, the focus will normally be passed to the main App window. This line ensures that control is returned to the application's terminal window.

Chapter 4
A Generic UIMS Application

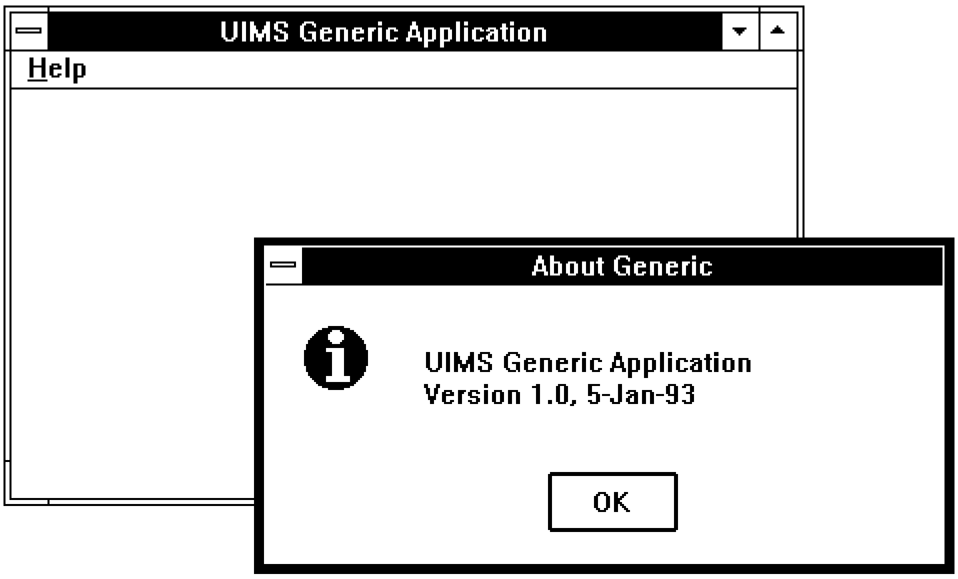
This chapter explains how to create a simple UIMS application called Generic which demonstrates the principles explained in Chapter 2. It covers the following topics:

- The essential parts of a UIMS application.
- Initialising and creating resources for a UIMS application.
- Writing the message loop.
- Terminating an application.
- The basic steps needed to build a UIMS application.

The Generic application will be used as the basis of all the sample applications in the chapters later in this manual.

Introduction

Generic is a standard Microsoft Windows application; that is, it meets the recommendations for user-interface style given in the IBM *System Application Architecture, Common User Access; Advanced Interface Design Guide*. Generic has a main window, a border, an application menu, and maximise and minimise boxes, but no other features. The application menu includes a Help menu with an About command, which, when chosen by the user, displays an About dialog box, describing Generic. The completed Generic application looks like Figure 4-1 when its About box is displayed.



UIMS Generic Application
Version 1.0, 5-Jan-93

Figure 4-1. The Generic Application

Generic is important, not for what it can do, but for what it provides: a template for writing UIMS applications. Building it helps you understand how UIMS applications are put together and how they work.

The Components of a UIMS Application

A UIMS application is any application that is specifically written to run with the REALITY User Interface Management System, and that uses the UIMS Application Programming Interface (API) to carry out its tasks. A UIMS application has the following basic components:

- An initialisation routine which defines the UIMS constants, sets up various variables and initialises UIMS.
- A sign on call to create an application context.
- A routine to create the resources needed by the application. This can be done dynamically, as part of the initialisation procedure, or can simply consist of loading a pre-defined resource file.
- A message loop to process messages relating to the application.
- A sign off call to destroy the application's resources and terminate the application.

Initialisation

The initialisation routine must contain two INCLUDE statements which integrate the UIMS constant definitions and variable assignments with the application, and a subroutine call to initialise UIMS.

```
INCLUDE UIMSDEFS FROM UIMS-TOOLS
INCLUDE UIMSCOMMON FROM UIMS-TOOLS
CALL InitialiseUims
```

It does not matter which INCLUDE statement appears first, but they must both come before the **InitialiseUims** call.

The initialisation routine can also include application-specific initialisation. In the case of the Generic application, the following line includes constant definitions which are common to both the DATA/BASIC source code and the resource script (see page 4-5):

```
INCLUDE GENERIC.H
```

These definitions must exist as an item in the REALITY file containing the DATA/BASIC source code, and also be available on the PC when compiling the resource script. The RealLink LanFTU file transfer facility can be used to copy the information from the host to the PC or vice versa.

Signing On and Off

Once you have initialised the UIMS for your application, you must sign on in order to create a application context. The context handle returned when you sign on must be passed to many of the UIMS subroutines and will also be needed when you sign off.

The **SignOn** subroutine requires two parameters: a string containing the name of the application and the name of a variable in which to return the application context. For example:

```
CALL SignOn("GENERIC", CONTEXT)
```

You should then test the context handle to check that the application is properly signed on. A context of zero indicates an error.

```
IF NOT(CONTEXT) THEN
  PRINT "Failed to Signon"
  STOP
END
```

The last action performed by your UIMS application must always be to sign off. This will destroy the application's resources and remove them from the screen. The **SignOff** subroutine requires two parameters: the application context handle returned when you signed on, and a variable in which to return the completion status.

```
CALL SignOff(CONTEXT, ERR)
STOP
```

Creating Resources

UIMS application resources consist of all the windows, menus and dialogue boxes presented to the user, together with their contents. These can be created in two ways:

- They can be pre-defined by preparing a resource script and compiling it using the UIMS Resource Compiler.
- They can be created dynamically during initialisation or as required by the application.

For the generic application, we will use a resource script. Note, however, that whichever method you use, you can modify your resources dynamically at any time, as required by your application.

The resources required by the Generic application consist of a main application window (AppWindow) and a menu bar with a single menu. The About box will be a message box with a single OK button; note that message boxes are displayed as they are created and the About box cannot therefore be defined in advance, but must be created when needed.

Creating a Resource Script

Resources are defined on the PC in a source file (resource script) which can be produced by any ASCII text editor (Windows Notepad, for example). The completed source must then be compiled, using the UIMS Resource Compiler, RLRC.

The following is the resource script for the Generic application:

```
#include GENERIC.H ❶

APPWINDOW = Win1 ❷
{
  TITLE = 'UIMS Generic Application'
  STYLE = CLOSABLE, SIZABLE, MOVABLE, ICONISABLE ❸
  BDRSTYLE = BORDER ❹
  POSITION = 125, 167 ❺
  SIZE = 500, 417 ❺
  MENUBAR = 0 ❻
  {
    MENU = 0 ❼
    {
      TITLE = '&Help' ❽
      CHILDREN = '&About Generic...'=HelpAbout ❾
    }
  }
}
```

❶ This line inserts the contents of a header file (GENERIC.H) which contains definitions of the constants used in the resource script and DATA/BASIC program (see page 4-19).

❷ This line introduces the definition for the application's main App window. The window is given the identifier Win1, which has been assigned a numeric value in the header file GENERIC.H.

❸ This line specifies the style of the window. It consists of a combination of a number of style elements. In this case the elements CLOSABLE, SIZABLE, MOVABLE and ICONISABLE are selected, producing a window that can be closed, moved, changed in size and minimised to an icon.

❹ The next line specifies whether or not the window has a border. It can have the value BORDER as shown above, or NONE to specify no border. Note however, that the window style controls the type of border and in some cases will automatically generate a border, even though none is specified. Refer to the *UIMS DATA/BASIC API*, *Reference Manual* for more details. In this case, the setting of the border style parameter is irrelevant because the SIZABLE style element automatically generates a double border. The parameter must be included, however, and setting it to BORDER will serve as a reminder that the window has a border.

❺ The next two lines specify the window's initial position on the screen and its size. These values will be interpreted as character or pixel units, depending on the current coordinate mode of the application context. If graphics mode is selected, the coordinates are interpreted as pixel positions on an arbitrary screen 1000 units wide by 1000 units high. When the resources are loaded into an application, the coordinates are scaled to fit within the actual screen.

The Generic application will run in graphics mode so the position is 125 units in from the left and 167 down, and the window size is 500 units wide by 417 high.

❻ This line introduces the definition for the App window's menu bar. Because the definition is nested within that of the App window, the menu bar will be automatically attached to the App window. The identifier of zero asks UIMS to assign a handle to this contact; this can be done in the case of the menu bar, because the application will not need access to this contact.

❼ This line introduces the definition for the Help menu. The nested definition makes the menu a child of the menu bar, and the identifier of zero asks UIMS to assign a handle to the contact.

❽ This line defines the title that will appear on the menu bar for the Help menu. The ampersand (&) specifies that the following letter (H) may be used in combination with the ALT key to select the menu from the keyboard.

❾ This line defines a menu item with the title About as a child of the Help menu. The menu item will have the identifier HelpAbout (which has been assigned a numeric value in the header file GENERIC.H), and can be selected from the menu by pressing the A key.

Loading the Compiled Resources

The compiled resources are loaded into a host UIMS application by calling the **LoadAppRes** subroutine. The following line loads the resources for the Generic application:

```
CALL LoadAppRes(CONTEXT❶, "generic.res"❷, ERR❸)
```

❶ This parameter is the handle of the application context.

❷ This is the name of the compiled resource file on the PC. If no path is specified, it is assumed to be in the resource directory specified in the RFW.INI file.

❸ This is a variable in which to return the completion status of the subroutine; a non-zero value indicates an error. Note, however, that if your application handles errors asynchronously (as it will, unless changed) the error parameter will always be returned set to zero, and any error will be reported in a Notify message. See page 4-12 for more details.

Displaying the Application Window

When the application window is complete, it can be displayed. This is done by adding the window as a child of the application context by calling the **AddChild** subroutine.

AddChild requires the following parameters:

- The handle of the application context.
- The handle of the parent object; that is the object to which the child is to be attached.
- A number indicating where, in any existing list of children, the new child is to be added. This allows, for instance, a new menu item to be added at the beginning of a menu rather than at the end.
- The handle of the child to be added.
- A variable in which to return any error code.

The following example attaches the application window to the context of the Generic application.

```
CALL AddChild(CONTEXT, CONTEXT, -1❶, Win1, ERR)
```

❶ The list index of -1 adds the new child at the end of any existing list.

If necessary, you can add several children in one operation by calling **AddChildren**. Refer to the *UIMS DATA/BASIC API, Reference Manual* for details of this subroutine.

Hiding the RealLink Window

Once you have displayed the App window, you can hide the RealLink window. This is not essential, and certainly should not be done until you have finished debugging your application, since any run time error messages will be displayed in the RealLink window.

To hide the RealLink window, include the following line in your application:

```
CALL SetTeWindow(0, 0, UIMS.NONE, ERR)
```

If your application hides the RealLink window, it must re-display it before returning to TCL. The following line re-displays the RealLink window:

```
CALL SetTeWindow(0, 0, TE.SHOWWIN, ERR)
```

The Message Loop

So far, all we have done is initialise our Generic application. The main part of the code consists of a loop where messages returned from the resources we have created are processed.

The loop has the following basic structure:

```
Until (user wants to exit) do
    Fetch the next message
    Process the message
Loop
```

Control of the loop can be achieved by using a variable which only changes value when the user selects the Close option from the system menu. For instance:

```
USER.WANTS.TO.EXIT = FALSE
LOOP UNTIL USER.WANTS.TO.EXIT DO

*   fetch message

*   if (user has selected Close)
        USER.WANTS.TO.EXIT = TRUE

*   else
*       process message

    end if

REPEAT
```

We will see later how we decide whether or not the user has selected Close.

Information about a UIMS event is obtained by using the **GetMsg** subroutine to fetch a message from the message queue. The subroutine requires ten parameters, as follows:

- A number representing how long (in tenths of a second) to wait for a message. This can allow an application to perform background tasks while waiting for messages. If zero is specified, **GetMsg** will not return until a message is received.

- Variables in which to return the handles of the application context, and the window and contact in which the event occurred.

- A variable in which to return the type of message.
- A variable in which to return a number representing the time the event occurred. This is only valid for certain types of event.
- Four variables in which to return additional message-specific parameters.

Message processing is best organised as a series of embedded case statements, with each level switching on a different message parameter. You are recommended to switch first on the window in which the event occurred, and then on the type of message. You can then, if necessary, test for the specific contact. It is unlikely that you will need to test the application context, as very few applications will have more than one.

The following example shows the code used to test messages received by the Generic application. Note that we are only interested in **UIMS.MSG.MENUITEM** and **UIMS.MSG.EXIT** messages and that all others are ignored, and that in this case the outer CASE structure is not really necessary, since the application has only one window. The subroutine HANDLE.WIN1.MENU tests for the selected menu item.

```
USER.WANTS.TO.EXIT = FALSE
LOOP UNTIL USER.WANTS.TO.EXIT DO

    CALL GetMsg(0❶, ...
                MSG.CONTEXT, ...
                MSG.WINDOW, ...
                MSG.CONTACT, ...
                MSG.TYPE, ...
                TIMESTAMP, ...
                DATA1, ...
                DATA2, ...
                DATA3, ...
                DATA4)

    BEGIN CASE
    CASE MSG.WINDOW=Win1

        BEGIN CASE

        CASE MSG.TYPE=UIMS.MSG.MENUITEM
            GOSUB HANDLE.WIN1.MENU

        CASE MSG.TYPE=UIMS.MSG.EXIT
            USER.WANTS.TO.EXIT = TRUE
        END CASE
    END CASE

REPEAT
    .
    .
    .
HANDLE.WIN1.MENUS:
    BEGIN CASE

    CASE MSG.CONTACT = HelpAbout
        GOSUB SHOW.ABOUT.BOX

    END CASE
RETURN
```

❶ This parameter signifies that **GetMsg** should not return until a message is received.

Displaying an About Box

The About box will consist of a message box with a single OK button. Message boxes are displayed as they are created and the About box cannot therefore be defined in advance, but must be created when needed.

A message box is created by calling the **CreateMessageBox** subroutine. This requires the following parameters:

- The handle of the application context.
- The type of message box required.
- The title of the message box.
- The message to be displayed.
- A list of button names (to replace the default names if required).
- A variable that will be used to return the selected button.
- A variable in which to return the completion status of the subroutine.

The example which follows creates the About box for the Generic application.

```
ABOUT.MESSAGE="UIMS Generic Application":CHAR(10):...
```

```
              "Version 1.0, 24-Mar-93" ❸

CALL CreateMessageBox(CONTEXT❶, ...
                      UIMS.INFO❷, ...
                      "About Generic"❸, ...
                      ABOUT.MESSAGE❹, ...
                      ""❺, ...
                      OK❻, ...
                      ERR❼)

IF ERR THEN
    ERRNO = ERR
    ERROR.STRING="Failed to create About message box"
    GOSUB ERROR.EXIT
END
```

❶ The first parameter is the handle of the application context, returned by the **SignOn** call.

❷ The second parameter specifies the message box type. **UIMS.INFO** specifies a box with an information icon and a single button. The button is labelled OK unless changed by providing a title in parameter ❺.

❸ The next parameter specifies the title of the message box.

❹ The next parameter specifies the message displayed in the box.

❺ The next parameter is a list of titles to use for the buttons. In this case, the parameter is a null string and the default title (OK) will therefore be used.

❻ The penultimate parameter is a variable in which the selected button in returned. The about box has only one button and the value returned will therefore, in this case, be ignored.

❼ The final parameter must be a variable in which a completion code can be returned. A return value of 0 indicates successful completion. Note that because **CreateMessageBox** returns a value (the selected button) as well as the completion code, no notify message will be generated, and this variable must be tested for successful completion as shown above.

Handling Errors

By default, UIMS handles errors asynchronously. This means that, where the only value returned by a UIMS subroutine is a completion code, this will always be returned set to **UIMS.SUCCESS** (zero), whether or not an error has occurred. If an error does occur, a **UIMS.MSG.NOTIFY** message will be generated, and this must be processed in the message loop. In the generic application, this is done as follows:

```
.
.
.
CASE MSG.WINDOW = 0 ❶

  BEGIN CASE
*
  CASE MSG.TYPE = UIMS.MSG.NOTIFY ❷

    ERRNO = DATA4 ❸
    ERROR.STRING = DATA2 ❹
    GOSUB ERROR.EXIT

  END CASE
    .
    .
    .
ERROR.EXIT:
  CALL GetErrorText(ERRNO, ERR.TEXT, ERR) ❺
  ERROR.STRING = ERROR.STRING:": ":CHAR(10):ERRNO:" - ":ERR.TEXT

  CALL CreateMessageBox(CONTEXT, ...
                        UIMS.INFO, ...
                        "Error", ...
                        ERROR.STRING, ...
                        "", ...
                        OK, ...
                        ERR)

  USER.WANTS.TO.EXIT = TRUE ❻
RETURN
```

❶ Notify messages are not associated with a particular window, so the event window handle is always zero.

❷ There are other types of message that have a zero window-handle, so we must specify Notify messages.

❸ The error number is returned in the *vData4* parameter.

❹ The *vData2* parameter returns a string with the name of the subroutine which failed.

❺ The **GetErrorText** subroutine returns a textual description of the specified error.

❻ Generic treats all UIMS errors as fatal. Setting this variable to **TRUE** causes the application to tidy up and return to TCL.

Note: If required, all errors can be handled synchronously. To do this, set UIMS into synchronous mode as follows.

```
CALL SetSync(CONTEXT, TRUE, ERR)
```

The second parameter must be **TRUE** to select synchronous mode, or **FALSE** for asynchronous.

Closing the Application

The user closes the application by selecting the Close option from the system menu. When this happens, a **UIMS.MSG.EXIT** message is generated. This must be tested for in the message loop so that housekeeping operations, such as saving open files, can be carried out before closing the application. In the Generic application, no housekeeping is necessary, so exiting is simply a matter of setting the USER.WANTS.TO.EXIT variable **TRUE** to cause the application to leave its message loop. The final actions are to re-display the RealLink window and then to sign off as described on page 4-4.

The Complete Application

The code for the complete Generic application is shown below. Note that in many cases the code fragments described above have been incorporated into short subroutines for ease of expansion and maintenance.

The DATA/BASIC Source

The DATA/BASIC source can be created on the host with a REALITY text editor (ED or SE). Alternatively it can be created on the PC with a text editor such as Windows Notepad, and then copied onto the host with one of the RealLink file transfer utilities (LanFTU or HOST-WS).

```
***************************************************************
*
* PROGRAM: GENERIC
*
* PURPOSE: Generic template for UIMS applications
*
* ROUTINES:
*   Main routine - initialises the application, loads resources,
*     processes messages
*   HANDLE.WIN1.MESSAGES - processes messages for Win1
*   HANDLE.WIN1.MENU - processes menu item messages for Win1
*   SHOW.ABOUT.BOX - displays an 'About' message box
*   ERROR.EXIT - processes errors
*
***************************************************************

* Definitions required for all UIMS applications
INCLUDE UIMSDEFS FROM UIMS-TOOLS
INCLUDE UIMSCOMMON FROM UIMS-TOOLS

* definitions specific to this application
INCLUDE GENERIC.H

* Sign on to UIMS

CALL InitialiseUims
CALL SignOn("GENERIC", CONTEXT)
IF NOT(CONTEXT) THEN
  PRINT "Failed to Signon"
  STOP
END
```

```
* Screen positions and contact sizes will be specified in pixels

CALL SetCoordMode(CONTEXT, UIMS.COORD.GRAPHIC, ERR)

* Load the resources
* The resource file must be on the PC, in the directory specified
* in the RFW.INI file

CALL LoadAppRes(CONTEXT, "generic.res", ERR)

* Add Win1 as a child of the context returned by the SignOn call.
* This has the effect of 'drawing' Win1 and its children.

CALL AddChild(CONTEXT, CONTEXT, -1, Win1, ERR)

* Hide the RealLink window
* This can be done at any time, but it's more reasuring to the
* user if we wait until the application's window has appeared.

CALL SetTeWindow(0, 0, UIMS.NONE, ERR)

* Message loop - continue until an EXIT message is received

USER.WANTS.TO.EXIT = FALSE
LOOP UNTIL USER.WANTS.TO.EXIT DO

* Fetch a message
* Note that GetMsg does not return until it has a message.

  CALL GetMsg(0, ...
              MSG.CONTEXT, ...
              MSG.WINDOW, ...
              MSG.CONTACT, ...
              MSG.TYPE, ...
              TIMESTAMP, ...
              DATA1, ...
              DATA2, ...
              DATA3, ...
              DATA4)

  BEGIN CASE ;* Switch on the window in which the event occurred.
*
```

```
  CASE MSG.WINDOW = Win1
    GOSUB HANDLE.WIN1.MESSAGES

  CASE MSG.WINDOW = 0

    BEGIN CASE
*
    CASE MSG.TYPE = UIMS.MSG.NOTIFY

      ERRNO = DATA4
      ERROR.STRING = DATA2
      GOSUB ERROR.EXIT

    END CASE

  END CASE

REPEAT

CALL SetTeWindow(0, 0, TE.SHOWWIN, ERR) ;* re-display the RealLink window
CALL SignOff(CONTEXT, ERR)              ;* sign off from UIMS

STOP                              ;* return to TCL


***************************************************************
*
* SUBROUTINE: HANDLE.WIN1.MESSAGES
*
* PURPOSE: Process messages for Win1
*
* COMMENTS:
*   This routine takes action according to the type of message.
*   In this application we only need to process menu item and exit
*   messages, but the routine is coded so that it can easily be
*   expanded to handle others as well.
*
***************************************************************

HANDLE.WIN1.MESSAGES:

    BEGIN CASE ;*   switch on the type of message
*
```

```
    CASE MSG.TYPE = UIMS.MSG.MENUITEM ;* user has selected a menu item
      GOSUB HANDLE.WIN1.MENU

    CASE MSG.TYPE = UIMS.MSG.EXIT ;* close the application
      USER.WANTS.TO.EXIT = TRUE

    END CASE

RETURN


***************************************************************
*
* SUBROUTINE: HANDLE.WIN1.MENU
*
* PURPOSE: Process Win1 menu item messages
*
* COMMENTS:
*   This routine takes action according to which menu item was
*   selected by the user. In this case we only have one menu,
*   with just one item, but the routine is coded so that it can
*   easily be expanded to handle more.
*
***************************************************************

HANDLE.WIN1.MENU:
  BEGIN CASE ;* Switch on the contact in which the event occurred
*
  CASE MSG.CONTACT=HelpAbout ;* About item on Help menu
    GOSUB SHOW.ABOUT.BOX

  END CASE

RETURN


***************************************************************
*
* SUBROUTINE: SHOW.ABOUT.BOX
*
* PURPOSE: Display a message giving details of the application
*
```

```
* COMMENTS:
*   The message is displayed in an About box - a UIMS information
*   box. This is application modal and will not allow the user to
*   do anything other than acknowledge the message.
*
***************************************************************

SHOW.ABOUT.BOX:
  * Set up the message
  * CHAR(10) starts a new line
  ABOUT.MESSAGE="UIMS Generic Application":CHAR(10):...
                "Version 1.0, 24-Mar-93"

  * Create a message box with an information icon and an OK button
  CALL CreateMessageBox(CONTEXT, ...
                        UIMS.INFO, ...
                        "About Generic", ...
                        ABOUT.MESSAGE, ...
                        "", ...
                        OK, ...
                        ERR)

  * If no error has occurred, the user must have operated the OK button
  IF ERR THEN
    ERRNO = ERR
    ERROR.STRING = "Failed to create About message box"
    GOSUB ERROR.EXIT
  END

RETURN


***************************************************************
*
* ROUTINE: ERROR.EXIT
*
* PURPOSE: Error handling routine
*
* COMMENTS:
*   Displays a message giving the type of error and where it
*   occurred, and then closes down the application.
*
***************************************************************
```

```
ERROR.EXIT:

* Get the text associated with the error code

  CALL GetErrorText(ERRNO, ERR.TEXT, ERR)
  ERROR.STRING = ERROR.STRING:": ":CHAR(10):ERRNO:" - ":ERR.TEXT

* Display it in a message box

  CALL CreateMessageBox(CONTEXT, ...
                        UIMS.INFO, ...
                        "Error", ...
                        ERROR.STRING, ...
                        "", ...
                        OK, ...
                        ERR)

  USER.WANTS.TO.EXIT = TRUE ;* close the application
RETURN

***************************************************************
END
```

Header File

The header file contains constant definitions which are common to both the DATA/BASIC source code and the resource script. These definitions must exist as an item in the REALITY file containing the source of the Generic application, and also be available on the PC when compiling the resource script. The RealLink LanFTU file transfer facility can be used to copy the information from the host to the PC or vice versa.

The header file for Generic must contain the following:

```
***************************************************************
*
* GENERIC.H - Constant definitions for Generic application
*
***************************************************************

EQUATE Win1 TO 10
EQUATE HelpAbout TO 20
```

Resource Script

The resource script must be created on the PC and given a name with the extension '.UCL'. It contains the definitions of the main App window, its menu bar, and the Help menu and About menu item:

```
**************************************************************
*
* GENERIC.UCL - Resource file for GENERIC program
*
**************************************************************

#include GENERIC.H

APPWINDOW = Win1
{
  TITLE = 'UIMS Generic Application'
  STYLE = CLOSABLE, SIZABLE, MOVABLE, ICONISABLE
  BDRSTYLE = BORDER
  POSITION = 125, 167
  SIZE = 500, 417
  MENUBAR = 0
  {
    MENU = 0
    {
      TITLE = '&Help'
      CHILDREN = '&About Generic...'=HelpAbout
    }
  }
}
```

Compiling the Generic Application

When you have created all the source files, you must carry out the following before you can run the Generic application:

1. Compile the resource script using the RLRC resource compiler on the PC. Make sure that the header file, GENERIC.H, is available on the PC; if necessary, copy it to the PC from the host.

   `RLRC` *path*`\GENERIC.UCL`

2. Copy the resulting resource file (GENERIC.RES) to the resource directory specified in RFW.INI.

3. Compile the DATA/BASIC source code on the host. Make sure that your source file includes an item containing the Generic header information (GENERIC.H); if necessary, copy it to the host from the PC.

   `BASIC` *FileName* `GENERIC`

4. Catalog the compiled program:

   `CATALOG` *FileName* `GENERIC`

You can now run the Generic program by entering:

```
GENERIC
```

Using Generic as a Template

Generic provides the essentials that make it an appropriate starting point for your UIMS applications. It conforms to the appearance standards given in the IBM *System Application Architecture, Common User Interface: Advanced Interface Design Guide*. The About Generic… command on the Help menu is included, and this displays an About dialog box, an application standard.

You can use Generic and the other examples in this manual as templates to build your own applications. To do this, copy and rename the sources of an existing application; then change any references to other source files within the copied files and insert new code. All sample applications in this manual have been created by inserting new code into copies of Generic's source files.

The following procedure explains how to use Generic as a template and adapt its source files to a new application:

1. Choose a name for your new application.

2. Copy the following REALITY file items, renaming them to match that of your new application: GENERIC, GENERIC.H.

3. Copy the following PC files, renaming them to match that of your new application: GENERIC.UCL, GENERIC.H.

   Note: The same header file should be used on both the host and the PC. You should create and edit this file on one or the other and then use the RealLink file transfer utilities to copy it as required.

4. Use a text editor to change the source code of your new application, replacing any references to Generic with the name of your new application. In particular, change the following:

   - The header file name: GENERIC.H.
   - The application name in the SignOn call.
   - The resource file name: GENERIC.RES.
   - The title and message in the About box.

5. Use a text editor to change the resource script of your new application, replacing any references to Generic with the name of your new application. In particular, change the following:

   - The header file name: GENERIC.H.

   - The title of the App window: 'UIMS Generic Application'.

   - The name of the HelpAbout menu item: 'About Generic...'.

Chapter 5
Windows

This chapter covers the following topics:

- Types of Window
- Creating a Window.
- Controlling the appearance of a Window.
- Positioning a Window on the screen.
- Output to a Window.

It also explains how to create a simple application, Output, that illustrates some of these concepts.

Types of Window

App and Child Windows

UIMS provides two main types of window: App (application) and Child windows.

- An App window can be displayed anywhere on the screen. It can therefore only have the application context as its parent. The main or root window of an application is always an App Window. An App window can have a menu bar with menus containing the application's commands

- A Child window is restricted to the client area of its parent. It cannot therefore have the application context as its parent, but it can be the child of an App window or another Child window. It can even be the child of a dialog box or an inclusive group. A Child window cannot have a menu bar.

In all other respects, App and Child windows are identical.

Other types of Window

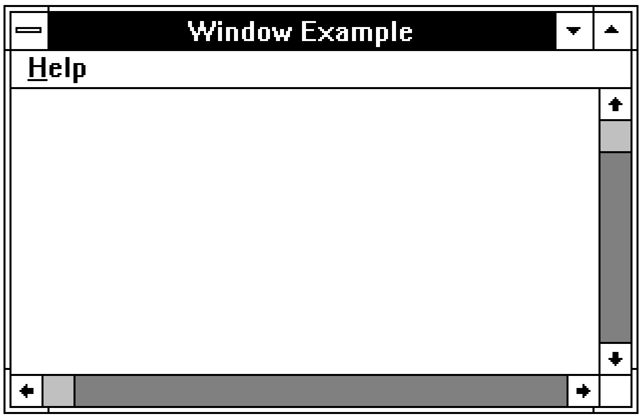
There are various other types of window; dialog boxes and inclusive groups for example. These do not have the same range of features as App and Child windows and are therefore described in the appropriate sections later in this manual.

Creating a Window

You can create a window either in your resource script, or by calling the appropriate create subroutine: **CreateAppWin** or **CreateChildWin**. You must provide the following information:

- The handle of the application context.
- A number by which the newly created window will be identified.
- A window title (if required).
- The position (relative to its parent) and size of the window. Note that the position will be that of the top left-hand corner of the complete window, including (if appropriate) the border, title bar, menu bar and scroll bars.
- The required styles for the window and its border.
- The handle of the window's parent.
- A variable in which to return the handle of the newly created window. The handle will normally be the same as the specified identifying number; however, if the identifier is zero, UIMS will assign a handle to the window.

The following examples both create an App window which has a title bar, a menu bar with a Help menu containing an About command, a system menu, and vertical and horizontal scroll bars. The window can be maximised, minimised, moved and changed in size.



In the resource script:

```
EQUATE Win1 TO 10
EQUATE Win1_Menus TO 11
EQUATE Win1_Help TO 20
EQUATE Help_About TO 21

APPWINDOW = Win1 ❶
{
    TITLE = 'Window Example' ❷
    STYLE = CLOSEABLE, SIZABLE, MOVABLE, ICONISABLE, VSCROLL,
              HSCROLL ❸
    BDRSTYLE = BORDER ❹
    SIZE = 300, 150 ❺
    POSITION = 30, 20 ❺

    MENUBAR = Win1_Menus ❻
    {
        MENU = Win1_Help
        {
            TITLE = '&Help'
            CHILDREN = '&About'=Help_About
        }
    }
}
```

❶ The APPWINDOW statement introduces the definition of an **AppWindow** contact. In this case, this has been given the identifier Win1, which is defined as having the value 10 at the beginning of the example.

❷ This line defines the title of the window. The text between the single quotes will appear in the title bar of the window.

❸ This line specifies the style of the window. In this case the window will have a system menu, maximise and minimise boxes, and vertical and horizontal scroll bars. It will also be possible to close, move and change the size of the window.

❹ The style of the window border is BORDER, giving the window a single border. Note, however, that in this case the SIZABLE window style makes the border double so that the size of the window can be changed with the mouse.

❺ The next two lines specify the initial size of the window and its position relative to its parent; in this case the window will be 500 coordinate units wide by 300 high and positioned 30 coordinate units across and 20 down. As this is an App window, the position is relative to the top left-hand corner of the screen. Note that the values given for the size and position assume that the application will be running in graphics mode (see page 5-10).

❻ The remainder of the example consists of definitions for the menu bar and the help menu. These are described in detail in Chapter 8.

While your application is running:

```
EQUATE Win1 TO 10
EQUATE Win1.Mbar TO 11
EQUATE Help.Menu TO 20
EQUATE Help.About TO 21

TITLE = "Window Example" ❶
STYLE = UIMS.WIN.CLOSEABLE + UIMS.WIN.SIZABLE + UIMS.WIN.MOVABLE ...
       + UIMS.WIN.ICONISABLE + UIMS.WIN.VSCROLL + UIMS.WIN.HSCROLL ❶
BDRSTYLE =  UIMS_BORDER ❶
CALL CreateAppWin(CONTEXT❷, Win1❸, TITLE, 30❹, 20❹, 500❺, ...
                    300❺, STYLE, BORDERSTYLE, CONTEXT❻, WIN1❼)

CALL CreateMenuBar(CONTEXT, Win1.Mbar, 0, WIN1.MBAR) ❽

CALL CreatePullDownMenu(CONTEXT, Help.Menu, "&Help", WIN1.MBAR, ...
                          HELP.MENU)
CALL CreateMenuItem(CONTEXT, Help.About, "&About", HELP.MENU, ...
                      HELP.ABOUT)
CALL AppWinSetMenuBar(CONTEXT, WIN1, WIN1.MBAR, ERR)
```

❶ These lines define variables containing the window title, style and border style, which will be used in the call to **CreateAppWin**.

❷ This is a variable containing the handle of the application context, obtained by a call to the **SignOn** subroutine.

❸ The second parameter to **CreateAppWin** is the identifier to be assigned to the contact. In this case, this has been given the identifier Win1, which is defined as having the value 10 at the beginning of the example.

❹ These two parameters specify the position of the window relative to its parent: 30 coordinate units across and 20 down. In this case the parent is the application context, so the position is relative to the screen.

❺ These two parameters specify the size of the window; in this case 500 coordinate units wide by 300 high.

❻ This parameter must be the handle of the window's parent; in this case the application context.

❼ This parameter is a variable in which to return the handle of the newly created window.

❽ The rest of the example creates a menu bar with a Help menu containing an About command. These are then attached to the **AppWindow** contact with **AppWinSetMenuBar**. Menu bars and menus are described in detail in Chapter 8.

Enabling and Disabling a Window

If a particular window is not appropriate in the current state of your application you can disable it. Disabled items do not respond to mouse clicks or keyboard selection.

To disable a window, call the **SetEnabled** subroutine and specify **FALSE** as the required state. For example:

```
CALL SetEnabled(Context.Handle, Window.Handle, FALSE, ERR)
```

To re-enable a disabled window, simply call **SetEnabled** with the second parameter set to **TRUE**:

```
CALL SetEnabled(Context.Handle, Window.Handle, TRUE, ERR)
```

If you prefer, you can use the **Disable** and **Enable** subroutines instead of **SetEnabled**.

The **GetState** subroutine returns whether a contact is enabled or disabled, and also whether it is mappable or unmappable.

Making a Window Visible

Many of the windows and other contacts that you create in your UIMS application will only be needed when you select particular commands. There are three ways in which you can control when a contact appears:

- You could create the contact only when it is needed and destroy it again when you have finished.

- You could create the contact as an orphan and attach it to its parent window when needed. When you have finished with the contact you would remove it from its parent's list of children.

- You could attach the contact to its parent, but set it unmappable (by calling the **SetMapped** subroutine) so that it will not normally be displayed. When you need the contact you simply change it to mappable, changing it back again when you have finished. Note that as alternatives to **SetMapped**, you could use **Map** and **UnMap**.

The last of these is the simplest and quickest method to use.

The **GetState** subroutine returns whether a contact is mappable or unmappable, and also whether it is enabled or disabled.

Destroying a Window

You can destroy a window by calling the **Destroy** subroutine. This deletes any internal record of the window and removes it from its parent's client area. The following example shows how this is done:

```
CALL Destroy(Context.Handle, Window.Handle, ERR)
```

Notes:

1. UIMS automatically destroys a window when its parent is destroyed.

2. If you destroy an application's root window this will have the effect of making the application invisible.

Controlling the Appearance of a Window

App and Child windows are the most complex objects which are available in UIMS. They each have over twenty different attributes, the most commonly used of which are described below.

Window Style

The style of a window is made up of a number of different style elements. As shown in the examples in the previous section, the style elements are added together to make up the style you require.

If you require, you can change the style of a window while your application is running, by calling **AppWinSetStyle** or **ChildWinSetStyle**. If you need to find out the current style of a window, you can call **AppWinGetStyle** or **ChildWinGetStyle** as appropriate. The following example changes the style of a Child window by adding the ability to minimise it.

```
CALL ChildWinGetStyle(CONTEXT❶, CHILDWIN❷, STYLE❸)
CALL BitTest(STYLE, UIMS.WIN.ICONISABLE, ICONISABLE) ❹
IF NOT(ICONISABLE) THEN
    STYLE = STYLE + UIMS.WIN.ICONISABLE
    CALL ChildWinSetStyle(CONTEXT❶, CHILDWIN❷, STYLE, ERR❺)
END
```

❶ This parameter is the handle of the application context.

❷ This parameter is the handle of the window.

❸ This parameter is a variable in which the current window style will be returned.

❹ This line determines whether the window can already be minimised. If not, the minimise style element (**UIMS.WIN.ICONISABLE**) is added to the current style. The style of the window is then changed using **ChildWinSetStyle**.

❺ This parameter is a variable in which a completion code will be returned.

Border Style

Another way in which you can change the appearance of a window is by specifying whether or not it has a border. This is set when the window is created, but can be changed with **SetBorderStyle**. You can find out the current style of a window's border by calling **GetBorderStyle**.

Note, however, that many of the window styles generate a border, overriding the border style setting. For example, a window with style **UIMS.WIN.SIZABLE** always has a double border, so that the user can change its size with the mouse.

Window Title

In many applications you will want to tell the user the current status by changing the window title. For instance, in a word processor you might display the name of the file which is being edited. You can change the title of an App window with **AppWinSetTitle**; a Child window cannot have a title.

Menu Bar

If you want your App window to have a menu bar, you must create this and its menus separately (see Chapter 8 for details). Once the menu bar has been created it can be attached to the window by calling **AppWinSetMenuBar**. There are also subroutines which allow you to remove the menu bar and to obtain the handle of the currently attached menu bar.

Note that you cannot attach a menu bar to a Child window.

Scroll Bars

In many cases you will need to give the user access to much more information than can be displayed in the client area of even a maximised window. A word processor, for example, will allow documents many pages long to be edited, but the client area of the application window can normally show only half a page at the most. UIMS allows you to add vertical and horizontal scroll bars to a window so that the user can choose which portion of the information will be displayed.

Scroll bars form part of the window style. They can be added to a window when it is created, by specifying the HSCROLL (**UIMS.WIN.HSCROLL**) and VSCROLL (**UIMS.WIN.VSCROLL**) style elements. These style elements can also be added or removed at any time by changing the window style with **AppWinSetStyle** or **ChildWinSetStyle**.

Scroll bars attached to a window are used in a similar way to scroll bar controls (described in Chapter 9). You will need to set values for the maximum and minimum positions, and the line and page movement increments. To do this, use **AppWinGetHScroll**, **AppWinGetVScroll**, **ChildWinGetHScroll** or **ChildWinGetVScroll**, as appropriate, to obtain the handle of the required scroll bar; you can then call **ScrollBarSetRange** and **ScrollBarSetInc**.

Setting Colours

The foreground and background colours of a window's client area can be set by changing the **Drawrule** object attached to the window. Note that any object to which a **Drawrule** can be attached initially inherits this from its parent. If you want to change the colours of a window without affecting its parent, you must create a new **Drawrule** to use.

Drawrules are described in more detail on page 5-14.

Positioning a Window on the Screen

When you create an App or Child window, you must specify its size and its position relative to its parent. Once created, it can be moved and resized as required.

The Coordinate System

UIMS has two screen coordinate systems: text and graphics. In text mode you specify the positions and sizes of windows and other contacts as a number of characters; for example, in text mode a window 76 coordinate units wide by 23 high will occupy most of the screen. Graphics mode gives you much finer control, since positions and sizes are specified in pixels; a graphics mode window the same size as in the previous example might be 608 coordinate units wide by 299 high.

Unless changed, UIMS applications use text mode. If you want to use graphics mode, call the **SetCoordMode** subroutine. You can use **GetCoordMode** to find out the current coordinate mode, while other subroutines allow you to find out the size of the screen in pixels, and the width and height in pixels of a screen font.

Note that in some cases you will not be able to use text mode – for instance, a standard width scroll bar is between two and three characters wide; if you want to create a standard width scroll bar contact, you must use graphics mode (refer to Chapter 9 for details).

All examples given in this manual assume that the application is using graphics mode.

In both coordinate modes, the top, left-hand corner of a contact's client area is position 0,0 (parent-relative coordinates). Note, however, that when positioning an App window (which has the application context as its parent) this origin is the top left-hand corner of the screen (screen-relative coordinates).

Moving and Sizing a Window

To move a window within its parent, call the **Move** subroutine. This positions the top left-hand corner of the window relative to the top left-hand corner of its parent's client area. The following example illustrates this.

`CALL Move(`*Context.Handle*`,` *Window.Handle*`, 35, 72, ERR)`

This moves the specified window to a position 35 coordinate units to the left of, and 72 coordinate units down from the top left-hand corner of its parent's client area.

Note: UIMS automatically moves any children when it moves the parent window.

You can find out the position of a window relative to its parent by calling the **GetPosition** subroutine.

If you want to change the size of a window, call the **Resize** subroutine. The following example makes a window 350 coordinate units wide by 220 units high.

```
CALL Resize(Context.Handle, Window.Handle, 350, 220, ERR)
```

Note that the size you specify is that of the complete window, including the border, title bar, etc.

Maximising and Minimising a Window

App windows with the styles **UIMS.WIN.SIZABLE** and **UIMS.WIN.ICONISABLE** can respectively be maximised and minimised by the user. There are also UIMS subroutines that allow you to perform these operations from within your application.

The **AppWinSetSizing** subroutine allows you to maximise and minimise an App window, and also to restore it to its previous size. Alternatively, you can use the separate maximise, minimise and restore subroutines (**AppWinMaximize**, **AppWinMinimize** and **AppWinRestore**).

Controlling Updates

Normally, any changes that affect the appearance of a window will take place immediately. Under some circumstances, however, you might prefer to make several changes and then display the combined result, without letting the user see the intermediate stages.

Updates can be controlled by the **SetUpdate** and **Draw** subroutines. There are two update modes: **UIMS.IMMEDIATE**, where updates take place immediately; and **UIMS.NONE**, where the window must be explicitly updated by calling the **Draw** subroutine.

GetUpdate returns the current update mode.

Output to a Window

There are two ways in which text and graphics can be displayed in the client area of a window: contacts can be created and made children of the window, or the UIMS drawing subroutines can be used to draw images directly on the client area. Each of these has its advantages and disadvantages.

- Contacts displayed within the client area are managed by UIMS and are automatically redrawn when necessary (for instance, if the window is minimised and restored). They are best used for static images – ones that are a constant feature of the window, and which do not move if the Window is resized or scrolled.
- Images drawn with the drawing subroutines must be managed by the application. For example, if the window is minimised and then restored, the application must redraw any images that were displayed in the Window immediately before it was minimised. This type of image is best used for information that is constantly changing – for the text of a document, for example, or for an image which is being edited in a graphics application.

Update Messages

UIMS tells the application that the client area of a window requires redrawing by generating an Update message. This typically occurs when the window is displayed, or when another application which is obscuring part or all of the window is moved. The message parameters indicate the position and size of the area that needs to be redrawn. Note that if this area is irregularly shaped (L-shaped, for instance), it is divided up into a number of rectangles and an Update message is generated for each.

The following example indicates the way in which update messages should be handled.

```
     .
     .
     .
CALL GetMsg(0, MSG.CONTEXT, MSG.WINDOW, MSG.CONTACT, MSG.TYPE, ...
              TIMESTAMP, DATA1, DATA2, DATA3, DATA4)

BEGIN CASE ;* Switch on the window in which the event occurred.
*
CASE MSG.WINDOW=Win1

  BEGIN CASE ;*   Now switch on the type of message
*
  CASE MSG.TYPE=UIMS.MSG.UPDATE
    GOSUB HANDLE.WIN1.UPDATES
```

.
.
.

The four GetMsg data parameters return the size and position of the region to be redrawn, relative to the top left-hand corner of the window's client area.

Note: The default UIMS event mask disables **UIMS.MSG.UPDATE** messages. If your application needs to process these messages, they can be enabled with the **SetEventMask** subroutine, as shown below:

```
CALL GetEventMask(CONTEXT, CONTEXT, EVENTMASK) ❶
CALL BitTest(EVENTMASK, UIMS.EM.UPDATE, ENABLED) ❷
IF NOT(ENABLED) THEN
  EVENTMASK = EVENTMASK + UIMS.EM.UPDATE ❸
  CALL SetEventMask(CONTEXT, CONTEXT, EVENTMASK, ERR) ❸
END
```

❶ The **GetEventMask** subroutine returns the current event mask for a specified contact. In this case, the second parameter specifies the application context. The third parameter is a variable in which the event mask is returned.

❷ The **BitTest** subroutine allows you to test an individual element in the mask. The third parameter is a variable in which the state of the specified element is returned.

❸ If update messages are not already enabled, they are added to the event mask and the new event mask is set by calling **SetEventMask**. As in the case of **GetEventMask**, the second parameter specifies the application context.

Using a Text Canvas

Under some circumstances, UIMS can reduce the amount of work required to keep the client area updated. A special style is available for App and Child windows which creates a text canvas, in which text drawn using the **DrawTextString** subroutine is stored. Whenever the client area needs redrawing, UIMS uses the stored information to restore the previously displayed text.

Note, however, that although a text canvas can reduce the application's work load, it has certain limitations.

- The text canvas stores only the text strings and their positions within the client area. The appearance of the text is determined by the **Font** object attached to the window; if the font is changed, the appearance of the text will change when it is next redrawn.

- Graphics shapes are not stored in the text canvas. The application must ensure that these are redrawn when the window is updated.

A text canvas is created by including the style **UIMS.WIN.TEXT** when creating the window. Once created, the canvas remains attached until the window is destroyed.

Drawing Tools

UIMS provides a number of objects which affect the appearance of text and graphics displayed in a window.

Pen	This controls the colour and width of lines drawn using the **Line** and **Rectangle** contacts, and the **DrawLine** and **DrawRect** subroutines.
Brush	This determines the way in which areas of a window client area are filled. It controls the colour used to fill rectangles drawn using the **Rectangle** contact and the **DrawRect** subroutine.
Font	This determines the characteristics of the text font used when writing characters on a window's client area. It controls the typeface used, and the style (bold, italic, etc.) and size of the characters.
Drawrule	This encapsulates the methods for drawing text and graphics in a window's client area. It determines which **Pen**, **Brush** and **Font** objects are used, and also controls the colours used for the window background and for text displayed in the window, and the way in which new text and graphics combine with that already displayed.

When a new contact is created and attached to a parent object, it is given the same Drawrule as its parent. UIMS provides default **Drawrule**, **Pen**, **Brush** and **Font** objects, which are inherited by contacts which have the application context as their parent. If you want a particular contact to have a different appearance, you can create a new Drawrule (and Pen, Brush and Font if required) and attach it to the contact concerned with the **SetDrawrule** subroutine. For example:

```
* Create a new Drawrule, using the default colours
FOREGROUND = UIMS.DEFAULT
BACKGROUND = UIMS.DEFAULT
TEXTMODE = UIMS.TEXT.OPAQUE
DRAWMODE = UIMS.DRAW.COPY
CALL CreateDrawrule(CONTEXT, NewDRule, FOREGROUND, BACKGROUND, ...
                      DRAWMODE, TEXTMODE, NEWDRULE)
```

```
* Create a red Pen, 3 pixels wide
COLOUR = UIMS.RED
WIDTH = 3
RESERVED = 0
CALL CreateDrawPen(CONTEXT, NewPen, COLOUR, WIDTH, RESERVED, NEWPEN)

* Attach the new Pen to the new Drawrule
CALL DrawruleSetPen(CONTEXT, NewDRule, NewPen, ERR)

* Now attach the new Drawrule to the required contact
CALL SetDrawrule(CONTEXT, CONTACT, NewDRule, ERR)
```

Drawing and Writing

UIMS provides only a limited range of drawing and writing functions, allowing you to draw straight lines, rectangles and text.

You draw straight lines with the **DrawLine** subroutine. The colour and width of the line are determined by the Pen object attached to the window's Drawrule. The following example draws a line from the point (10, 90) to the point (360, 90)

```
CALL DrawLine(CONTEXT, CONTACT, 10, 90, 360, 90, 0❶, ERR)
```

❶ This parameter is for future use. It must be set to a numeric value, but its value will be ignored.

The **DrawRect** subroutine allows you to draw rectangles. The width and colour of the border is determined by the Pen object attached to the window's Drawrule, while the colour of the interior is set by the selected Brush object. The following example draws a rectangle which has its top-left and bottom-right corners at positions (10, 30) and (60, 80) respectively.

```
CALL DrawRect(CONTEXT, CONTACT, 10, 30, 60, 80, 0❶, ERR)
```

❶ This parameter is for future use. It must be set to a numeric value, but its value will be ignored.

You can display text by calling the **DrawTextString** subroutine. The Font object attached to the window's Drawrule determines the style of the text, while the Drawrule itself specifies its colour. The following example displays the string "Some Sample Text" at the point (1, 1).

```
CALL DrawTextString(CONTEXT, CONTACT, "Some Sample Text", 1, 1, ERR)
```

Controlling the Client Area

In many cases, the text and graphics displayed within a window will need to change as the user operates the application. UIMS provides two subroutines, **Scroll** and **Erase**, to help you control the appearance of the client area.

The **Scroll** subroutine allows you to move the contents of the client area relative to its containing window. You can scroll up, down, left or right, and can specify whether all or only part of the client area should be scrolled. The following example scrolls the whole of the client area up by 13 pixels. Note that text and graphics within 13 pixels of the top of the client area will be lost and that the bottom 13 pixels of the client area will be erased; that is, filled with the background colour specified by the window's Drawrule.

Note: In addition to scrolling the window, the Scroll subroutine generates an update message, specifying the erased region of the client area. Any images to be scrolled into the client area should be displayed in response to this message.

```
CALL Scroll(CONTEXT, WINDOW, 0❶, 13❷, 0❸, 0❸, 0❸, 0❸, ERR)
```

❶ This parameter specifies the amount of horizontal movement. Since, in this case, we are scrolling up, it is set to zero.

❷ This parameter specifies the amount of vertical movement – 13 pixels.

❸ These four parameters specify the region of the client area to be scrolled. In this case, all are set to zero, specifying the whole of the client area. If only part of the client area was to be scrolled, these would define the left, top, right and bottom edges of the scrolled region.

The **Erase** subroutine allows you to erase all or part of the client area, filling the erased area with the background colour specified by the window's Drawrule. The following example erases a rectangle which has its top-left and bottom-right corners at positions (25, 15) and (75, 90) respectively.

```
CALL Erase(CONTEXT, CONTACT, 25, 15, 75, 90, ERR)
```

An Example Application: Output

The example application, Output, illustrates how to use the **UIMS.MSG.UPDATE** message to restore the client area, and demonstrates the differences between drawing directly into the client area, and using a text canvas and graphics objects. It does this by creating two Child windows, one maintained by UIMS and the other by the application.

Output is an extension of the Generic application described in Chapter 4. To create the Output application, copy and rename the source files of the Generic application and then make the following modifications.

1. Add new constant definitions.
2. Define ChildWindow, Drawrule, Brush and Pen resources.
3. Enable update and create messages.
4. Add a UIMS.MSG.CREATE case to the message loop.
5. Add a UIMS.MSG.UPDATE case to the message loop.
6. Add a UIMS.MSG.DESTROY case to the message loop.
7. Modify the UIMS.MSG.EXIT case.
8. Modify the ERROR.EXIT subroutine.
9. Compile the resource file and the DATA/BASIC program.

Add New Constant Definitions

You will need identifiers for the additional resources defined in the resource script. These must be available to both the resource script and the DATA/BASIC source, so add the following to your header file.

```
EQUATE Child1 TO 30
EQUATE Child2 TO 40

EQUATE BlueRect TO 50
EQUATE YellowLine TO 60

EQUATE Drawrule1 TO 100
EQUATE RedBrush TO 110
EQUATE BlueBrush TO 111
EQUATE GreenPen TO 120
EQUATE YellowPen TO 121
```

Ensure that the new header file is available on both the host and the PC.

Define New Resources

You must create the drawing tools to be used in Output's client area before any drawing is carried out. Since you need to create these tools only once, the best place to do this is in the resource script. Add the following lines to the file OUTPUT.UCL before the definition of Win1:

```
DRAWRULE = Drawrule1 /* used by the graphic objects */
{
  FOREGROUND = BLACK
  BACKGROUND = WHITE
  DRAWMODE = COPY
  TEXTMODE = OPAQUE
  BRUSH = BlueBrush
  {
    STYLE = SOLID
    FOREGROUND = BLUE
  }
  PEN = YellowPen
  {
    STYLE = SOLID
    FOREGROUND = YELLOW
    WIDTH = 3
  }
}

BRUSH = RedBrush /* used for rectangle in Child2 */
{
  STYLE = SOLID
  FOREGROUND = RED
}

PEN = GreenPen /* used for diaglonal line in Child2 */
{
  STYLE = SOLID
  FOREGROUND = GREEN
  WIDTH = 3
}

CHILDWINDOW = Child1 /* the left-hand half of Win1 */
{
  BDRSTYLE = NONE
  POSITION = 0, 0
  SIZE = 450, 500
  STYLE = TEXT         /* give it a text canvas */

  RECTANGLE = BlueRect /* rectangle object */
  {
    STYLE = NONE
    STARTPOS = 0, 0      /* arbitrary initial position */
    ENDPOS = 100, 100    /* arbitrary initial size */
    DRAWRULE = Drawrule1 /* specifies the blue brush */
    MAPPED = FALSE       /* hide it until needed */
  }

  LINE = YellowLine /* line object */
  {
    ENDSTYLE = NONE
    STARTPOS = 0, 0      /* arbitrary initial position */
    ENDPOS = 100, 100    /* arbitrary initial size */
    DRAWRULE = Drawrule1 /* specifies the yellow pen */
    MAPPED = FALSE       /* hide it until needed */
  }
}

CHILDWINDOW = Child2 /* the right-hand half of Win1 */
{
  BDRSTYLE = NONE
  POSITION = 450, 0
  SIZE = 450, 500
  STYLE = NONE       /* no text canvas */
}
```

Then add the following line to the definition of Win1:

```
  CHILDREN = Child1, Child2
```

Enable Update Messages

When you start your application update messages will be disabled. To enable them, add the following code to the DATA/BASIC source after signing on to UIMS, but before loading the resources.

```
CALL GetEventMask(CONTEXT, CONTEXT, EVENTMASK)
CALL BitTest(EVENTMASK, UIMS.EM.UPDATE, ENABLED)
IF NOT(ENABLED) THEN EVENTMASK = EVENTMASK + UIMS.EM.UPDATE
```

```
CALL SetEventMask(CONTEXT, CONTEXT, EVENTMASK, ERR)
```

Enable Create Messages

Create messages are not controlled through the normal event mask mechanism, but must be separately enabled by calling the **SetSecondaryEventMask** subroutine. Add the following line immediately after the code which enables Update messages:

```
CALL SetSecondaryEventMask(CONTEXT, EVENTMASK, TRUE, FALSE, ERR)
```

Note that this also enables UIMS.MSG.DESTROY messages.

Add the Create Case

The **UIMS.MSG.CREATE** message informs your application that an App window has been created. In the Output application, this will be used to draw text and display objects in the left-hand Child window, Child1.

To handle the **UIMS.MSG.CREATE** message, add the following CASE statement to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.CREATE
  GOSUB HANDLE.WIN1.CREATE
```

The following is the HANDLE.WIN1.CREATE subroutine called by this CASE statement. This can be added to the source code at any convenient point.

```
***************************************************************
*
* SUBROUTINE: HANDLE.WIN1.CREATE
*
* PURPOSE: Process the Win1 create message
*
* COMMENTS:
*   Draws some text in the default font and then positions,
*   sizes and maps two graphics contacts: a blue rectangle and
*   a diagonal yellow line. The Rectangle and Line contacts are
*   are loaded from the resource file.
*
***************************************************************

HANDLE.WIN1.CREATE:
  * Get the size characteristics of the font.
  * This will be used to determine the vertical spacing of the
  * text in the window.
  * The window's font is that attached to the window's Drawrule.
  CALL GetDrawrule(CONTEXT, Child1, DEFAULT.DRAWRULE)
  CALL DrawruleGetFont(CONTEXT, DEFAULT.DRAWRULE, DEFAULT.FONT)
  CALL FontGetMetrics(CONTEXT, ...
                      DEFAULT.FONT, ...
                      HEIGHT, ...
                      ASCENT, ...
                      DESCENT, ...
                      LEADING, ...
                      LCWIDTH, ...
                      UCWIDTH, ...
                      MAXWIDTH, ...
                      ERR)

  * The vertical spacing will be standard height (HEIGHT) of the
  * selected font, plus the standard amount of spacing between
  * adjacent lines (LEADING).

  VSPACE = HEIGHT + LEADING

  * Initialise the drawing position to one line from the top of
  * the screen and one standard upper case character from the
  * left-hand edge.

  HPOS = UCWIDTH
  VPOS = VSPACE

  * Send characters to the screen. After displaying each line of
  * text, advance the vertical position by the vertical spacing
  * determined above, ready for the next line of text.

  TEXT = "These characters are drawn onto the Text"
  CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)
  VPOS = VPOS + VSPACE

  TEXT = "Canvas and UIMS therefore manages"
  CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)
  VPOS = VPOS + VSPACE

  TEXT = "updating the display. The yellow line"
  CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)
  VPOS = VPOS + VSPACE

  TEXT = "and blue rectangle below are UIMS"
  CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)
```

```
  VPOS = VPOS + VSPACE

  TEXT = "objects, and redraw themselves when"
  CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)
  VPOS = VPOS + VSPACE

  TEXT = "necessary."
  CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)
  VPOS = VPOS + VSPACE

  * Move print position 5 characters to the right and 2 lines down
  HPOS = HPOS + (5 * UCWIDTH)
  VPOS = VPOS + (2 * VSPACE)

  * Blue rectangle
  CALL Move(CONTEXT, BlueRect, HPOS, VPOS, ERR)
  CALL Resize(CONTEXT, BlueRect, 150, 60, ERR)
  CALL Map(CONTEXT, BlueRect, ERR)

  * Yellow line
  CALL Move(CONTEXT, YellowLine, HPOS, VPOS, ERR)
  CALL Resize(CONTEXT, YellowLine, 150, 60, ERR)
  CALL Map(CONTEXT, YellowLine, ERR)

RETURN
```

Add the Update Case

The **UIMS.MSG.UPDATE** message informs your application when it should redraw all or part of its client area. In the Output application, this will be used to draw text and graphics in the right-hand Child window, Child2. In many ways, the routine is similar to the HANDLE.WIN1.CREATE subroutine shown above.

To handle the **UIMS.MSG.UPDATE** message, add the following CASE statement to the main message case statement:

```
  CASE MSG.WINDOW = Child2
    GOSUB HANDLE.CHILD2.MESSAGES
```

The following is the HANDLE.CHILD2.MESSAGES subroutine called by this CASE statement. This can be added to the source code at any convenient point.

```
***************************************************************
*
* SUBROUTINE: HANDLE.CHILD2.MESSAGES
```

```
*
* PURPOSE: Process messages for the Child2 window
*
* COMMENTS:
*   Draws some text in the default font and then draws a red
*   rectangle with a green line across it. The Brush and Pen
*   contacts required are loaded from the resource file.
*
***************************************************************

HANDLE.CHILD2.MESSAGES:
  BEGIN CASE
  CASE MSG.TYPE = UIMS.MSG.UPDATE

    * The window font has not been changed so it is the same as
    * that of Child1. We can therefore use the position and
    * spacing values set up when Win1 was created

    VSPACE = HEIGHT + LEADING

    * Initialise the drawing position to one line from the top of
    * the screen and one standard upper case character from the
    * left-hand edge.

    HPOS = UCWIDTH
    VPOS = VSPACE

    * Send characters to the screen. After displaying each line of
    * text, advance the vertical position by the vertical spacing
    * determined above, ready for the next line of text.

    TEXT = "These characters, and the red rectangle"
    CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)
    VPOS = VPOS + VSPACE

    TEXT = "and green line below, are drawn directly"
    CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)
    VPOS = VPOS + VSPACE

    TEXT = "onto the client area. The application"
    CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)
    VPOS = VPOS + VSPACE
    TEXT = "must therefore redraw the display each"
    CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)
    VPOS = VPOS + VSPACE

    TEXT = "time an Update message is received."
    CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)
    VPOS = VPOS + VSPACE

    * Move the print position 5 characters to the right
    * and 2 lines down
    HPOS = HPOS + (5 * UCWIDTH)
    VPOS = VPOS + (2 * VSPACE)

    * Draw a red rectangle
    CALL DrawruleGetBrush(CONTEXT, DEFAULT.DRAWRULE, OLDBRUSH)
    CALL DrawruleSetBrush(CONTEXT, DEFAULT.DRAWRULE, RedBrush, ERR)
    CALL DrawRect(CONTEXT, Child2, HPOS, VPOS, HPOS+150, ...
                    VPOS+60, UIMS.NONE, ERR)
    CALL DrawruleSetBrush(CONTEXT, DEFAULT.DRAWRULE, OLDBRUSH, ERR)

    * Draw a green line
    CALL DrawruleGetPen(CONTEXT, DEFAULT.DRAWRULE, OLDPEN)
    CALL DrawruleSetPen(CONTEXT, DEFAULT.DRAWRULE, GreenPen, ERR)
    CALL DrawLine(CONTEXT, Child2, HPOS, VPOS, HPOS+150, ...
                    VPOS+60, 0, ERR)
    CALL DrawruleSetPen(CONTEXT, DEFAULT.DRAWRULE, OLDPEN, ERR)

  END CASE

RETURN
```

Add the Destroy Case

In enabling **UIMS.MSG.CREATE** messages, you have also enabled **UIMS.MSG.DESTROY** messages. You must therefore modify your message loop to process these messages. Add the following to the HANDLE.WIN1.MESSAGES subroutine:

```
    CASE MSG.TYPE = UIMS.MSG.DESTROY
      USER.WANTS.TO.EXIT = TRUE
```

Modify the Exit Case

Since the USER.WANTS.TO.EXIT variable is now set when a Destroy message is received, the Exit case must initiate the generation of this message by destroying the App window. Find the lines in the HANDLE.WIN1.MESSAGES subroutine which read:

```
    CASE MSG.TYPE = UIMS.MSG.EXIT ;* close the application
      USER.WANTS.TO.EXIT = TRUE
```

and change them to read:

```
    CASE MSG.TYPE = UIMS.MSG.EXIT ;* close the application
      CALL Destroy(CONTEXT, Win1, ERR)
```

Modify the Error Subroutine

In Generic, the ERROR.EXIT routine sets the USER.WANTS.TO.EXIT variable, but in Output it must destroy the App window and thus generate a Destroy message. Find the line in the ERROR.EXIT subroutine which reads:

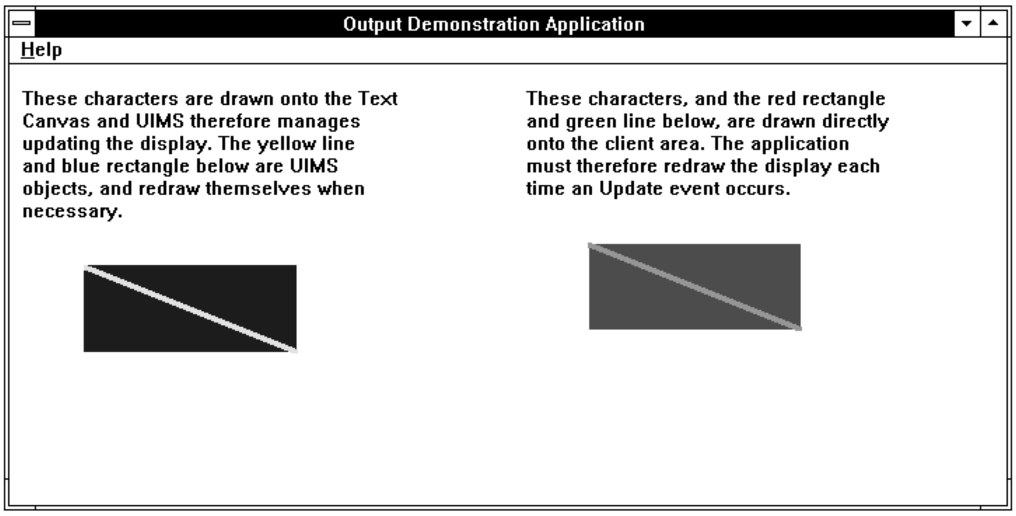
```
  USER.WANTS.TO.EXIT = TRUE
```

and change it to read:

```
  CALL Destroy(CONTEXT, Win1, ERR)
```

Compile

When you have made these changes, you can compile the resource script and DATA/BASIC program as described in Chapter 4 for the Generic application. When run, the application should look like this:



These characters are drawn onto the Text Canvas and UIMS therefore manages updating the display. The yellow line and blue rectangle below are UIMS objects, and redraw themselves when necessary.

These characters, and the red rectangle and green line below, are drawn directly onto the client area. The application must therefore redraw the display each time an Update event occurs.

Figure 5-1. The Output Application

As you use Output and other applications, you will see the two halves of the App window redrawn. Note that the left-hand half of the window, with its text canvas and graphics contacts, will always be redrawn faster than the right-hand half. This is because updates to Child1 are handled entirely by UIMS and do not involve communication with the host.

Chapter 6
Keyboard and Mouse Input

Most applications require input from the user. Typically, input from the user comes via the keyboard or the mouse. In UIMS, applications receive keyboard and mouse input in the form of input messages.

This chapter covers the following topics:

- The input messages that UIMS sends your application.
- Responding to UIMS input messages.

It also explains how to create an example application that responds to various types of input message.

Input Messages

Whenever the user presses a key, moves the mouse or clicks a mouse button, UIMS responds by sending input messages to the active application. UIMS also sends input messages in response to Timer input.

UIMS provides several types of input message:

Keyboard	User input through the keyboard.
Mouse	User input through the mouse.
Timer	Input from the system timer.
Scroll-bar	User input through a window's scroll-bars.
Menu	User input through a window's menus.

The keyboard, mouse and timer messages correspond directly to hardware input. The scroll-bar and menu messages are generated in response to mouse and keyboard actions outside the client area of the window.

Keyboard Input

Much of an application's input comes from the keyboard. Whenever the user presses a key, a **UIMS.MSG.KEYPRESS** message is generated.

The *vData2* parameter of a **UIMS.MSG.KEYPRESS** message contains the UIMS virtual key code (key alias) of the key that was pressed. A virtual key code is a device-independent value for a specific key or key-combination. UIMS uses virtual key codes so that it can provide consistent keyboard input, no matter what computer your application is running on.

The *vData1* parameter contains the states of the various modifier keys (SHIFT, CTRL, ALT, etc.).

An application receives keyboard messages only when it has 'input focus'. Your application receives input focus when it is the active application; that is, when the user has selected your application's window. If required, you can use the **SetContactFocus** subroutine to set the focus to a particular window within your application. The **GetChildFocus** subroutine can be used to determine which window has the focus.

Mouse Input

User input can also come from the mouse. UIMS sends mouse messages to an application when the user moves the mouse pointer into and through any of its windows, or presses or releases a mouse button while the pointer is in any of its windows. UIMS generates mouse messages in response to the following events:

UIMS.MSG.MOTION
The user has moved the pointer into or through the window.

UIMS.MSG.PRESS
The user has pressed a mouse button.

UIMS.MSG.RELEASE
The user has released a mouse button.

UIMS.MSG.CLICK
The user has clicked (pressed and released) a mouse button.

UIMS.MSG.DBLCLICK
The user has double-clicked (pressed and released twice) a mouse button.

UIMS.MSG.DRAG
The user has started or stopped dragging the mouse; that is, moving the mouse while holding down its primary button (button 1).

For all these types of message, the *vData1* and *vData2* parameters contain the horizontal and vertical coordinates of the pointer position. *vData3* contains the states of the keyboard modifier keys and of any mouse buttons that have not changed state. *vData4* contains the number of any mouse button that has changed state (been pressed or released).

UIMS sends mouse messages to a window only if the pointer is in the window, or if you have captured mouse input by calling **GrabPointer**. This subroutine directs UIMS to send all mouse input to a specified window, regardless of the position of the mouse pointer. Applications should typically use this subroutine to take control of the mouse when carrying out some critical operation with the mouse, such as selecting something in the client area. Capturing the mouse prevents other applications taking control of the mouse before the operation is completed.

Since the mouse is a shared resource, it is important to release the captured mouse as soon as you have finished the operation. The mouse is released by calling the **UngrabPointer** subroutine.

Note: If a contact's event mask has enabled pointer drag messages, when a drag event starts within that contact, UIMS will automatically perform a **GrabPointer**, followed by an **UngrabPointer** when the drag ends.

Timer Input

UIMS sends timer input to your application when the specified interval elapses for a particular timer. To receive timer input, you must create a timer by calling the **AddTimer** subroutine. Once created, a timer runs continuously, generating a **UIMS.MSG.TIMER** message every time its period expires.

The following example shows how to create a timer for a 5 second interval:

```
CALL AddTimer(CONTEXT, 5000❶, TIM5SEC❷)
```

❶ This parameter sets an interval of 5000 milliseconds. This means that the timer will generate a **UIMS.MSG.TIMER** message every 5 seconds.

❷ This parameter is a variable in which the handle of the timer will be returned. This can be used to distinguish between messages generated by different timers, and also to remove the timer when it is no longer required (by calling the **RemoveTimer** subroutine).

Scroll-bar Input

When the user operates a window's scroll-bar (with either the mouse or the keyboard), UIMS generates a scroll-bar input message. The message type is either **UIMS.MSG.HSCROLL** or **UIMS.MSG.VSCROLL** depending on whether the scroll-bar is aligned horizontally or vertically.

Applications use the scroll-bar messages to direct scrolling within the window. Applications that display text or other data that is too large to fit in the client area usually provide some form of scrolling. Scroll-bars are an easy way to let the user direct scrolling actions.

To get scroll-bar input, add scroll-bars to the window. You can do this by including the **UIMS.WIN.HSCROLL** and **UIMS.WIN.VSCROLL** style elements when you create the window. The following example creates scroll-bars for the window Win1:

```
TITLE = "Input Example Application"
STYLE = UIMS.WIN.CLOSABLE + UIMS.WIN.ICONISABLE + UIMS.WIN.MOVABLE +
UIMS.WIN.SIZABLE + UIMS.WIN.HSCROLL + UIMS.WIN.VSCROLL
BORDER = UIMS.BORDER
CALL CreateAppWin(CONTEXT, Win1, TITLE, 20, 20, 300, 200, STYLE,
BORDER, CONTEXT, WIN1)
```

UIMS displays the scroll-bars when it displays the window. It automatically maintains the scroll-bars and sends scroll-bar messages to the application when the user moves the scroll-bar thumb.

When the application receives a scroll-bar message, the *vData4* parameter indicates the new thumb position and *vData2* specifies the type of scrolling requested, as listed below:

UIMS.SB.LEFT	The user clicked the scroll-bar Left arrow.
UIMS.SB.RIGHT	The user clicked the scroll-bar Right arrow.
UIMS.SB.UP	The user clicked the scroll-bar Up arrow.
UIMS.SB.DOWN	The user clicked the scroll-bar Down arrow.
UIMS.SB.PAGELEFT	The user clicked the scroll-bar thumb-track to the left of the thumb.
UIMS.SB.PAGERIGHT	The user clicked the scroll-bar thumb-track to the right of the thumb.
UIMS.SB.PAGEUP	The user clicked the scroll-bar thumb-track above the thumb.
UIMS.SB.PAGEDOWN	The user clicked the scroll-bar thumb-track below the thumb.
UIMS.SB.THUMB	The user has stopped dragging the thumb.
UIMS.SB.THUMBTRACK	The user is dragging the thumb.

Menu Input

Whenever the user chooses a command from one of the application's menus, UIMS generates a **UIMS.MSG.MENUITEM** message. Refer to Chapter 8 for details of how to use menus and menu input.

An Example Application: Input

The example application, Input, illustrates how to process input messages, using the keyboard, the mouse, a timer and scroll bars as examples. It displays the current or most recent state of each of these input mechanisms. To create the Input application, copy and rename the source files of the Generic application, as described in Chapter 4. Then make the following modifications:

1. Modify the style of the App Window.
2. Enable the messages you intend processing.
3. Set up the positions at which the states of the input mechanisms will be displayed.
4. Initialise new variables.
5. Add the UIMS.MSG.CREATE case.
6. Add the UIMS.MSG.DESTROY case.
7. Modify the UIMS.MSG.EXIT case.
8. Add the UIMS.MSG.KEYPRESS case.
9. Add the UIMS.MSG.MOTION case.
10. Add the UIMS.MSG.CLICK and UIMS.MSG.DBLCLICK cases.
11. Add the UIMS.MSG.HSCROLL and UIMS.MSG.VSCROLL cases.
12. Add the UIMS.MSG.TIMER case.
13. Modify the ERROR.EXIT subroutine.
14. Compile the resource file and the DATA/BASIC program.

Although Windows does not require a pointing device, this example assumes that you have a mouse or other pointing device. If you do not have a mouse, the application will not receive mouse input messages.

Modify the Window Style

You will need to modify the definition of the application's App window so that a window with horizontal and vertical scroll bars will be created. In addition, you should give the window a text canvas so that you do not have to process Update messages. Change the definition of Win1 in the resource script so that the STYLE parameter looks like this:

```
STYLE = CLOSABLE,
        SIZABLE,
        MOVABLE,
        ICONISABLE,
        HSCROLL,        /* scroll bars */
        VSCROLL,
        TEXT            /* text canvas */
```

Enable Messages

Of the types of message you require, only Keyboard messages are enabled when you start a UIMS application. To enable Mouse, Timer and Scroll messages, add the following code to the DATA/BASIC source after signing on to UIMS, but before loading the resources.

```
CALL GetEventMask(CONTEXT, CONTEXT, EVENTMASK)
EVENTMASK = EVENTMASK + UIMS.EM.MOTION + UIMS.EM.CLICK ...
              + UIMS.EM.DBLCLICK
EVENTMASK = EVENTMASK + UIMS.EM.TIMER
EVENTMASK = EVENTMASK + UIMS.EM.HSCROLL + UIMS.EM.VSCROLL
CALL SetEventMask(CONTEXT, CONTEXT, EVENTMASK, ERR)
```

In addition, you will need to enable Create messages. This is done by the following subroutine call, which must follow the code given above:

```
CALL SetSecondaryEventMask(CONTEXT, EVENTMASK, TRUE, FALSE, ERR)
```

Note that this also enables **UIMS.MSG.DESTROY** messages.

Set the Text Positions

The vertical spacing of the text in the window will be determined by the size characteristics of the window's font. The following lines of code set the positions at which the different messages are displayed.

```
CALL GetDrawrule(CONTEXT, Win1, WIN1.DRAWRULE)
CALL DrawruleGetFont(CONTEXT, WIN1.DRAWRULE, WIN1.FONT)
CALL FontGetMetrics(CONTEXT, WIN1.FONT, HEIGHT, ASCENT, DESCENT,
LEADING, LCWIDTH, UCWIDTH, MAXWIDTH, ERR)

VSPACE = HEIGHT + LEADING
```

```
HPOS = UCWIDTH ;* one upper case character width from the left edge
MOUSE.VPOS = VSPACE ;* one line from the top
BUTTON.VPOS = MOUSE.VPOS + VSPACE
KEY.VPOS = BUTTON.VPOS + VSPACE
SCROLL.VPOS = KEY.VPOS + VSPACE
TIMER.VPOS = SCROLL.VPOS + VSPACE
```

This code must be included after the application's resources have been created, but before the start of the message loop.

Initialise Variables

The text strings that will be displayed in the window will be set up in the appropriate case statements. They must, however, be initialised before they can be used. Add the following lines near the beginning of the application:

```
KEYTEXT = ""
MOUSETEXT = ""
BUTTONTEXT = ""
SCROLLTEXT = ""
TIMERTEXT = ""
```

Add the Create Case

The **AddTimer** subroutine creates a timer and sets it running. Since you only need to do this once, the UIMS.MSG.CREATE case is a convenient place to do this. Add the following to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.CREATE
  CALL AddTimer(CONTEXT, 5000, TIMER)
  TIMER.COUNT = 0
```

Add the Destroy Case

In enabling **UIMS.MSG.CREATE** messages, you have also enabled **UIMS.MSG.DESTROY** messages. You must therefore modify your message loop to process these messages. Add the following to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.DESTROY
  USER.WANTS.TO.EXIT = TRUE
```

Modify the Exit Case

In the Input application, the UIMS.MSG.EXIT case must stop the timer before the application terminates and, because the USER.WANTS.TO.EXIT variable is now set when a Destroy message is received, it must initiate the generation of this message by destroying the App window.

Find the lines in the HANDLE.WIN1.MESSAGES subroutine which read:

```
    CASE MSG.TYPE = UIMS.MSG.EXIT ;* close the application
      USER.WANTS.TO.EXIT = TRUE
```

and change them to read:

```
    CASE MSG.TYPE = UIMS.MSG.EXIT ;* close the application
      CALL RemoveTimer(CONTEXT, TIMER, ERR)
      CALL Destroy(CONTEXT, Win1, ERR)
```

Add the Key Press Case

Add a UIMS.MSG.KEYPRESS case to process keyboard operations. The following lines must be added to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.KEYPRESS
  CALL FontGetTextLen(CONTEXT, WIN1.FONT, KEYTEXT, LENGTH)
  CALL Erase(CONTEXT, Win1, HPOS, KEY.VPOS, HPOS+LENGTH, ...
               KEY.VPOS+HEIGHT, ERR)
  KEYTEXT = "UIMS.MSG.KEYPRESS: ":DATA2:", ":DATA1
  CALL DrawTextString(CONTEXT, Win1, KEYTEXT, HPOS, KEY.VPOS, ERR)
```

Add the Mouse Motion Case

Add a UIMS.MSG.MOTION case to process mouse-movement. The following lines must be added to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.MOTION
  CALL FontGetTextLen(CONTEXT, WIN1.FONT, MOUSETEXT, LENGTH)
  CALL Erase(CONTEXT, Win1, HPOS, MOUSE.VPOS, HPOS+LENGTH, ...
               MOUSE.VPOS+HEIGHT, ERR)
  MOUSETEXT = "UIMS.MSG.MOTION: ":INT(DATA1 / 65536):...
                 ", ":DATA2:", ":DATA3
  CALL DrawTextString(CONTEXT, Win1, MOUSETEXT, HPOS, ...
                        MOUSE.VPOS, ERR)
```

Add the Click and Double Click Cases

Add the UIMS.MSG.CLICK and UIMS.MSG.DBLCLICK cases to process mouse button operations. The following lines must be added to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.CLICK OR MSG.TYPE = UIMS.MSG.DBLCLICK
  CALL FontGetTextLen(CONTEXT, WIN1.FONT, BUTTONTEXT, LENGTH)
  CALL Erase(CONTEXT, Win1, HPOS, BUTTON.VPOS, HPOS+LENGTH, ...
               BUTTON.VPOS+HEIGHT, ERR)
  IF MSG.TYPE = UIMS.MSG.CLICK THEN
    BUTTONTEXT = "UIMS.MSG.CLICK: "
  END ELSE
    BUTTONTEXT = "UIMS.MSG.DBLCLICK: "
  END
  BUTTONTEXT = BUTTONTEXT:INT(DATA1 / 65536):", ":DATA2:", ":...
                DATA3:", ":DATA4
  CALL DrawTextString(CONTEXT, Win1, BUTTONTEXT, HPOS, ...
                       BUTTON.VPOS, ERR)
```

Add the Scroll Cases

Add the UIMS.MSG.HSCROLL and UIMS.MSG.VSCROLL cases to process scroll bar operations. The following lines must be added to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE (MSG.TYPE = UIMS.MSG.HSCROLL OR MSG.TYPE = UIMS.MSG.VSCROLL)
  CALL FontGetTextLen(CONTEXT, WIN1.FONT, SCROLLTEXT, LENGTH)
  CALL Erase(CONTEXT, Win1, HPOS, SCROLL.VPOS, HPOS+LENGTH, ...
             SCROLL.VPOS+HEIGHT, ERR)
  IF MSG.TYPE = UIMS.MSG.HSCROLL THEN
    SCROLLTEXT = "UIMS.MSG.HSCROLL: "
    BEGIN CASE
      CASE DATA2 = UIMS.SB.LEFT
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.LEFT, "
      CASE DATA2 = UIMS.SB.RIGHT
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.RIGHT, "
      CASE DATA2 = UIMS.SB.PAGELEFT
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.PAGELEFT, "
      CASE DATA2 = UIMS.SB.PAGERIGHT
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.PAGERIGHT, "
    END CASE
  END ELSE
    SCROLLTEXT = "UIMS.MSG.VSCROLL: "
    BEGIN CASE
      CASE DATA2 = UIMS.SB.UP
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.UP, "
      CASE DATA2 = UIMS.SB.DOWN
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.DOWN, "
      CASE DATA2 = UIMS.SB.PAGEUP
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.PAGEUP, "
      CASE DATA2 = UIMS.SB.PAGEDOWN
        SCROLLTEXT = SCROLLTEXT:"UIMS.SB.PAGEDOWN, "
    END CASE
  END

  BEGIN CASE
    CASE DATA2 = UIMS.SB.THUMB
      SCROLLTEXT = SCROLLTEXT:"UIMS.SB.THUMB, "
    CASE DATA2 = UIMS.SB.THUMBTRACK
      SCROLLTEXT = SCROLLTEXT:"UIMS.SB.THUMBTRACK, "
  END CASE
  SCROLLTEXT = SCROLLTEXT:DATA4

  CALL DrawTextString(CONTEXT, Win1, SCROLLTEXT, HPOS, ...
                      SCROLL.VPOS, ERR)
```

Add the Timer Case

For a **UIMS.MSG.TIMER** message, the event window parameter is zero. Find the lines in the message loop which read:

```
CASE MSG.WINDOW = 0
  BEGIN CASE
```

and add following lines after them:

```
CASE MSG.TYPE = UIMS.MSG.TIMER
  CALL FontGetTextLen(CONTEXT, WIN1.FONT, TIMERTEXT, LENGTH)
  CALL Erase(CONTEXT, Win1, HPOS, TIMER.VPOS, HPOS+LENGTH, ...
             TIMER.VPOS+HEIGHT, ERR)

  TIMER.COUNT = TIMER.COUNT + 5
  TIMERTEXT = "UIMS.MSG.TIMER: ":TIMER.COUNT:" seconds"
  CALL DrawTextString(CONTEXT, Win1, TIMERTEXT, HPOS, ...
                      TIMER.VPOS, ERR)
```

Modify the Error Subroutine

The ERROR.EXIT subroutine needs the same changes as the UIMS.MSG.EXIT case. Find the line in the ERROR.EXIT subroutine which reads:

```
  USER.WANTS.TO.EXIT = TRUE
```

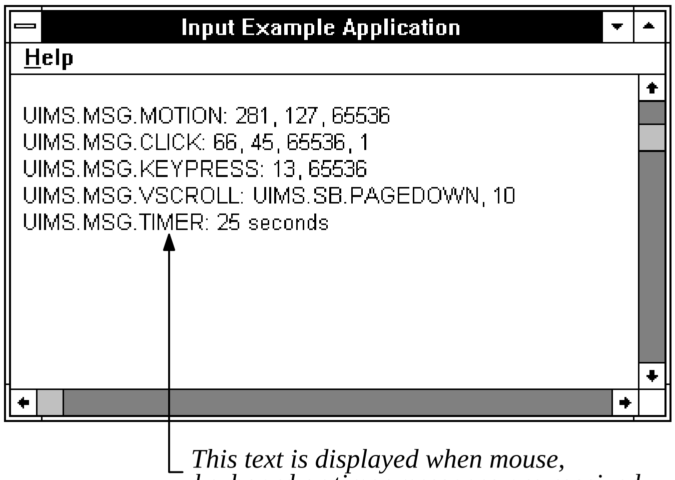
and replace it with the following:

```
  CALL RemoveTimer(CONTEXT, TIMER, ERR)
  CALL Destroy(CONTEXT, Win1, ERR)
```

Note that this is only necessary for errors that you are treating as fatal.

Compile

When you have made these changes, you can compile the resource script and DATA/BASIC program as described in Chapter 4 for the Generic application. Test the application by moving the mouse, clicking and double clicking the mouse buttons, pressing keys on the keyboard and using the scroll bars. The application should look like Figure 6-1.



```
UIMS.MSG.MOTION: 281, 127, 65536
UIMS.MSG.CLICK: 66, 45, 65536, 1
UIMS.MSG.KEYPRESS: 13, 65536
UIMS.MSG.VSCROLL: UIMS.SB.PAGEDOWN, 10
UIMS.MSG.TIMER: 25 seconds
```

This text is displayed when mouse, keyboard or timer messages are received.

Figure 6-1. The Input Application

Chapter 7
The Pointer, the Mouse and the Keyboard

The pointer is a special graphics image that shows the user where actions initiated by the mouse will take place. In most UIMS applications, the user makes selections, chooses commands, and directs other actions by using either the mouse or the keyboard.

This chapter covers the following topics:

- Displaying and controlling the shape of the pointer.
- Letting the user use the keyboard to move the pointer.
- Letting the user select information with the mouse.

It also explains how to create an example application that illustrates some of these concepts.

Displaying the Pointer

No one pointer shape can satisfy the all needs of an application. For example, a text editor or word processor would normally need an I-beam pointer when the user is editing text, but it would need an hourglass pointer when carrying out lengthy operations, such as reading a file from disk.

The shape of the pointer is controlled by attaching a **Pointer** object to each window. The pointer changes to the specified shape while it is within the client area of the window concerned.

When the window is created, it is given a default pointer which, unless changed, has the standard diagonal arrow shape. If you require a different shape in a particular window, you can create a new **Pointer** object with a different shape to attach to the window. A **Pointer** object can be created in your resource script, or within your application by calling the **CreatePointer** subroutine. The following examples illustrate these two methods:

In the resource script:

```
#define Win2 200
#define Win2Pointer 201

APPWINDOW = Win2
{
  .
  .  * definitions for Win2
  .

  POINTER = Win2Pointer
  {
    TYPE = IBEAM
  }
}
```

In your application:

```
* create a new pointer object with I-beam shape
CALL CreatePointer(CONTEXT, Win2Pointer, UIMS.PTR.IBEAM, ...
                     WIN2POINTER)

* attach it to the window
CALL SetPointer(CONTEXT, Win2, Win2Pointer, ERR)
```

Should you need to, there are two ways of changing the pointer shape while your application is running. The preferred method is to use the **SetPointer** subroutine to attach a different **Pointer** object to the window, as shown below:

```
* get the handle of the current pointer,
* so that it can be restored later
CALL GetPointer(CONTEXT, Win2, OLDPOINTER)

* now attach a cross-hair pointer
CALL SetPointer(CONTEXT, Win2, CrossPointer, ERR)

.
. ;* operations that need a cross-hair pointer
.

* restore the old pointer
CALL SetPointer(CONTEXT, Win2, OLDPOINTER, ERR)
```

Alternatively, you can change the type of an existing pointer by calling **PointerSetType**.

Displaying the Hourglass during a Lengthy Operation

Whenever your applicaton begins a lengthy operation, such as reading a large block of data to a disk file, you should change the shape of the pointer to the hourglass. This lets the user know that a lengthy operation is in progress and that they should wait before attempting to continue their work. Once the operation is complete, your application should restore the pointer to its previous shape.

If you wish, you can create a **Pointer** object with the hourglass shape and attach this to the window concerned. However, UIMS provides two subroutines, **WaitPointerOff** and **WaitPointerOn**, which simplfy this process. Both methods are shown in the following examples.

Using the wait-pointer subroutines:

```
.
.
.
* change the pointer to the hourglass
CALL WaitPointerOn(CONTEXT, ERR)
.
. ;* lengthy operation
.
* restore the previous pointer shape
```

```
CALL WaitPointerOff(CONTEXT, ERR)
.
.
.
```

Using a **Pointer** object:

```
CALL CreatePointer(CONTEXT, HourGlass, UIMS.PTR.WAIT, HOURGLASS) ❶
.
.
.
CALL GrabPointer(CONTEXT, Win1, ERR) ❷
CALL GetPointer(CONTEXT, Win1, OLDPOINTER) ❸
CALL SetPointer(CONTEXT, Win1, HourGlass, ERR) ❹
.
. ;* lengthy operation
.
CALL SetPointer(CONTEXT, Win1, OLDPOINTER, ERR) ❺
CALL UngrabPointer(ERR) ❻
.
.
.
```

❶ This line creates a new **Pointer** object and gives it the hour glass shape. Alternatively, this could be defined in the resource script.

❷ The application first captures the mouse input using the **GrabPointer** subroutine. This keeps the user from attempting to use the mouse to carry out work in another application while the lengthy operation is in progress. When the mouse input is captured, UIMS directs all mouse input messages to the specified window, regardless of whether the pointer is in that window. The application can then process the messages as appropriate.

❸ The handle of the previous **Pointer** object is saved for later restoration.

❹ The pointer is changed to the hour glass pointer.

❺ When the lengthy operation is complete, the application restores the previous **Pointer** object.

❻ The **UngrabPointer** subroutine releases the mouse input.

Using the Pointer with the Keyboard

Windows does not require a pointing device and applications should therefore provide the user with a way to duplicate mouse actions with the keyboard. To allow the user to move the pointer using the keyboard, use the **SetPointerPos**, **SetPointer** and **GetPointerPos** subroutines to display and move the pointer.

Using the Keyboard to Move the Pointer

You can use the **SetPointerPos** subroutine to move the pointer from within your application. This subroutine is typically used to let the user move the pointer using the keyboard.

To move the pointer, detect the **UIMS.MSG.KEYPRESS** message and filter for the virtual-key values of the direction keys: UIK.UP, UIK.DOWN, UIK.LEFT and UIK.RIGHT. On each keystroke, the application should update the position of the pointer. The following example shows how this might be done:

```
CASE MSG.TYPE = UIMS.MSG.SIZE ❶
   WIN1.WIDTH = INT(DATA1 / 65536)
   WIN1.HEIGHT = DATA2

CASE MSG.TYPE=UIMS.MSG.KEYPRESS

   IF DATA2 = UIK.DOWN ...
      OR DATA2 = UIK.UP ...
      OR DATA2 = UIK.LEFT ...
      OR DATA2 = UIK.RIGHT THEN ❷

      CALL GetPointerPos(CONTEXT, Win1, PT.HPOS, PT.VPOS, ERR) ❸

      BEGIN CASE

      * adjust the pointer position according to which key was
      * pressed

         CASE DATA2 = UIK.UP
            PT.VPOS = PT.VPOS - 1

         CASE DATA2 = UIK.DOWN
            PT.VPOS = PT.VPOS + 1

         CASE DATA2 = UIK.LEFT
            PT.HPOS = PT.HPOS - 1
         CASE DATA2 = UIK.RIGHT
            PT.HPOS = PT.HPOS + 1

      END CASE

      IF PT.HPOS >= WIN1.WIDTH THEN ❹
         PT.HPOS = WIN1.WIDTH - 1
      END ELSE
         IF PT.HPOS < 0 THEN PT.HPOS = 1
      END
      IF PT.VPOS >= WIN1.HEIGHT THEN
         PT.VPOS = WIN1.HEIGHT - 1
      END ELSE
         IF PT.VPOS < 0 THEN PT.VPOS = 1
      END

      CALL SetPointerPos(CONTEXT, Win1, PT.HPOS, PT.VPOS, ERR) ❺

   END

* End of UIMS.MSG.KEYPRESS case
```

In this example:

❶ Each time a **UIMS.MSG.SIZE** message is received, the current width and height of the window's client area are stored in the WIN1.WIDTH and WIN1.HEIGHT variables. Although there are other ways of obtaining this information, this is by far the simplest.

❷ The first IF statement filters for the virtual-key values of the direction keys: UIK.UP, UIK.DOWN, UIK.LEFT and UIK.RIGHT.

❸ The **GetPointerPos** subroutine returns the current position of the pointer. Since the second parameter identifies a specific window, the position returned will be relative to the client area of that window. Note that, if the mouse is available, the user could potentially move the pointer with the mouse at any time; there is therefore no guarantee that the position values saved on the last keystroke are correct.

The application uses client-relative coordinates for two reasons: mouse messages give the pointer position relative the the window's client area; and client-relative coordinates do not have to be updated if the window moves.

❹ These IF statements check that the new pointer position is within the client area. If necessary the position is adjusted.

❺ The **SetPointerPos** subroutine moves the pointer to the new location. The position is relative to the window specified in the the second parameter.

Letting the User Select Information with the Mouse

Many Windows applications (word processors, graphics editors, etc.) allow the user to select text and graphics with the mouse. Though this capability could be programmed into a UIMS application, the delays inherent in the communications link would make it very slow and difficult to use. There are, however, two UIMS contacts which allow text selection – the EditBox and the TextEditor. If you need to allow the user to select text, you should use one or more of these contacts, as described in Chapter 9. Chapter 11 shows you how to transfer text between these contacts and the clipboard.

An Example Application: Pointer

The example application, Pointer, illustrates how to move the pointer with the keyboard, and how to display the hourglass during a lengthy process. It also shows you how different windows can be given differently shaped pointers. The main application window contains four child windows, each of which has a different pointer object attached to it.

Pointer is an extension of the Generic application described in Chapter 4. To create the Pointer application, copy and rename the source files of the Generic application and then make the following modifications.

1. Add new constant definitions.
2. Define ChildWindow and Pointer resources.
3. Enable size messages.
4. Add a UIMS.MSG.SIZE case to the message loop.
5. Add a UIMS.MSG.KEYPRESS case to the message loop.

Add New Constant Definitions

You will need identifiers for the additional resources defined in the resource script. These must be available to both the resource script and the DATA/BASIC source, so add the following to your header file.

```
EQUATE Child1      TO 50
EQUATE Pointer1    TO 51
EQUATE Child2      TO 60
EQUATE Pointer2    TO 61
EQUATE Child3      TO 70
EQUATE Pointer3    TO 71
EQUATE Child4      TO 80
EQUATE Pointer4    TO 81
```

Ensure that the new header file is available on both the host and the PC.

Define New Resources

Pointer requires four Child windows, defined as children of the App window, and a Pointer object for each Child window. These can all be defined in the resource script. Add the following lines to the definition of Win1 in the file POINTER.UCL:

```
CHILDWINDOW = Child1
{
  POSITION = 50, 50
  SIZE = 100, 100
  BDRSTYLE = BORDER
  STYLE = NONE
  POINTER = Pointer1
  {
    TYPE = CROSS
    PATTERN = ''
    POSITION = 0, 0
  }
}

CHILDWINDOW = Child2
{
  POSITION = 300, 50
  SIZE = 100, 100
  BDRSTYLE = BORDER
  STYLE = NONE
  POINTER = Pointer2
  {
    TYPE = IBEAM
    PATTERN = ''
    POSITION = 0, 0
  }
}

CHILDWINDOW = Child3
{
  POSITION = 50, 200
  SIZE = 100, 100
  BDRSTYLE = BORDER
  STYLE = NONE
  POINTER = Pointer3
  {
    TYPE = PLUS
    PATTERN = ''
    POSITION = 0, 0
  }
}

CHILDWINDOW = Child4
{
  POSITION = 300, 200
  SIZE = 100, 100
  BDRSTYLE = BORDER
  STYLE = NONE
  POINTER = Pointer4
  {
    TYPE = PLUS
    PATTERN = ''
    POSITION = 0, 0
  }
}
```

Enable Size Messages

When you start your application, update messages will be disabled. To enable them add the following code to the DATA/BASIC source after signing on to UIMS, but before loading the resources.

```
* Enable size messages if not already enabled
CALL GetEventMask(CONTEXT, CONTEXT, EVENTMASK)
CALL BitTest(EVENTMASK, UIMS.EM.SIZE, ENABLED)
IF NOT(ENABLED) THEN EVENTMASK = EVENTMASK + UIMS.EM.SIZE
CALL SetEventMask(CONTEXT, CONTEXT, EVENTMASK, ERR)
```

Add the Size Case

Pointer keeps track of changes to the size of Win1 by monitoring Size messages. To handle the **UIMS.MSG.SIZE** message, add the following CASE statement to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.SIZE
  WIN1.WIDTH = INT(DATA1 / 65536)
  WIN1.HEIGHT = DATA2
```

Add the Keypress Case

The user must be able to use the keyboard to move the pointer and also to initiate a lengthy operation. This is done by processing **UIMS.MSG.KEYPRESS** messages, and testing the DATA2 variable to find out which key was pressed. The RETURN key is used to initiate a lengthy operation; when this is done, the Pointer application suspends processing for 10 seconds using the DATA/BASIC SLEEP statement. The cursor control keys (up, down, left and right) are used to move the pointer.

To handle the **UIMS.MSG.KEYPRESS** message, add the following CASE statement to the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.KEYPRESS
  GOSUB HANDLE.WIN1.KEYS
```

The following is the HANDLE.WIN1.KEYS subroutine called by this CASE statement. This can be added to the source code at any convenient point.

```
***************************************************************
*
* ROUTINE: HANDLE.WIN1.KEYS
*
* PURPOSE: Processes keypress messages for Win1
*
* COMMENTS:
*   The RETURN key initiates a long procedure (simulated by
*   sleeping for 10 seconds), during which the hourglass is
*   displayed.
*   The cursor keys move the mouse pointer within the client
*   area of Win1.
*
***************************************************************

HANDLE.WIN1.KEYS:
  BEGIN CASE
*
  * pressing the return key initiates a long procedure
  CASE DATA2 = UIK.RETURN
    * display the hourglass pointer
    CALL WaitPointerOn(CONTEXT, ERR)

    * simulate a long procedure
    CALL Erase(CONTEXT, Win1, 0, 0, 0, 0, ERR)
    CALL DrawTextString(CONTEXT, Win1, "Long procedure...", ...
           10, 10, ERR)
    SLEEP 10
    CALL DrawTextString(CONTEXT, Win1, ...
           "Procedure took 10 seconds to complete.", 10, 10, ERR)

    * return to the previous pointer
    CALL WaitPointerOff(CONTEXT, ERR)

  * the pointer can be moved with the cursor keys
  CASE DATA2 = UIK.DOWN OR ...
       DATA2 = UIK.UP OR ...
       DATA2 = UIK.LEFT OR ...
       DATA2 = UIK.RIGHT

    * get the current cursor position
    CALL GetPointerPos(CONTEXT, Win1, PT.HPOS, PT.VPOS, ERR)
    BEGIN CASE
     * adjust the pointer position
     * according to which key was pressed
*
    CASE DATA2 = UIK.UP
      PT.VPOS = PT.VPOS - 5

    CASE DATA2 = UIK.DOWN
      PT.VPOS = PT.VPOS + 5

    CASE DATA2 = UIK.LEFT
      PT.HPOS = PT.HPOS - 5

    CASE DATA2 = UIK.RIGHT
      PT.HPOS = PT.HPOS + 5

    END CASE

    * stop the pointer moving outside the client area
    IF PT.HPOS >= WIN1.WIDTH THEN
      PT.HPOS = WIN1.WIDTH - 1
    END ELSE
      IF PT.HPOS < 0 THEN PT.HPOS = 1
    END
    IF PT.VPOS >= WIN1.HEIGHT THEN
      PT.VPOS = WIN1.HEIGHT - 1
    END ELSE
      IF PT.VPOS < 0 THEN PT.VPOS = 1
    END

    * set the new pointer position
    CALL SetPointerPos(CONTEXT, Win1, PT.HPOS, PT.VPOS, ERR)

  END CASE
RETURN
```

Compile

When you have made these changes, you can compile the resource script and DATA/BASIC program as described in Chapter 4 for the Generic application. When run, the Pointer application should look like Figure 7-1.

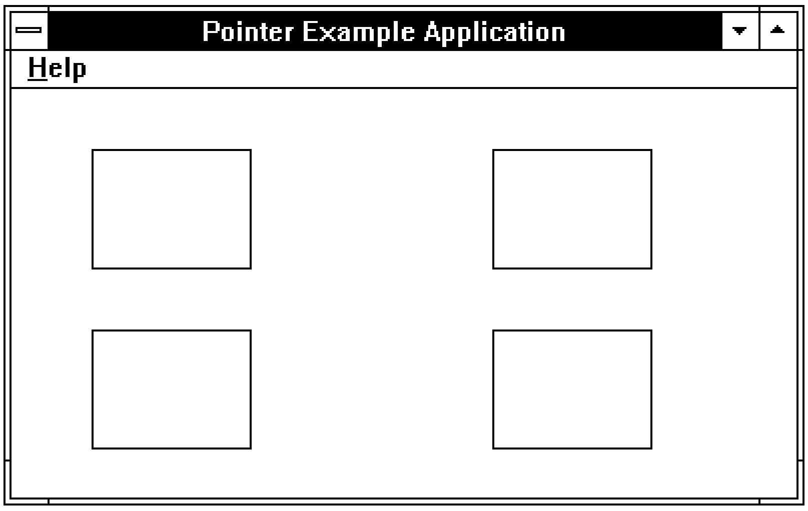


Figure 7-1. The Pointer Application

Each box is a Child window. As you move the mouse into a box, the pointer will change to a cross, I-beam or plus sign; moving out again restores the arrow shape.

You can also use the direction keys to move the pointer and press RETURN to start a lengthy operation; the pointer changes to the hourglass to indicate that a lengthy operation is in progress.

Chapter 8
Menus

Most UIMS applications use menus to let the user select commands or actions.

This chapter describes what a menu is and tells you how to:

- Define a menu and include it in your application.
- Process input from a menu.
- Modify an existing menu.
- Use special menu features.

An example application, EditMenu, which uses and processes input from menus is also described.

What is a Menu?

A menu is a list of items which, to the user, are the application's commands. The user tells the application to perform a command by using the mouse or the keyboard to select an item from a menu. When a user chooses a menu item, UIMS sends a message to the application that specifies which item the user selected.

The primary application menu is normally the menu bar along the top of the main application window. Each item on this menu can either allow the user to select commands directly or, more usually, offer a second, pull-down, menu with more commands. In some cases, an item on the pull-down menu will offer a further, cascading, menu.

Figure 8-1 illustrates these different kinds of menu.

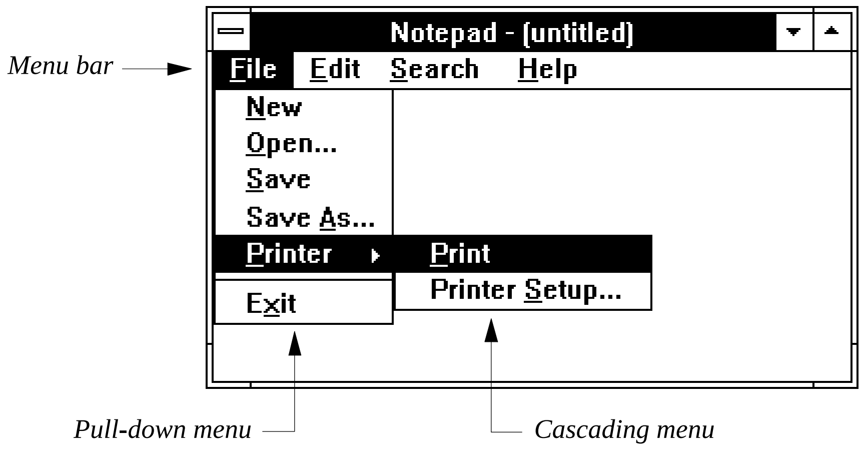


Figure 8-1. **Menus**

The menu items are the most important elements in this hierarchy, because they initiate commands within the application. The menu bar and pull-down menus are simply the means of presenting these commands to the user.

Defining a Menu

UIMS menus consist of three elements:

- A **MenuBar** contact which is attached to an Application Window.
- **Menu** contacts (pull-down menus) which are attached to the menu bar or, in some cases, to other menus (cascading menus).
- **MenuItem** contacts which are attached to the pull-down menus or, in some cases, directly to the menu bar.

All these elements must be defined, either within your application, or in a pre-defined resource script which is compiled in advance and loaded into your application at run time.

The simplest way of defining a menu is to define it in your application's resource script (.UCL) file. The definition of a menu must specify:

- The name of the **MenuBar** contact.
- The names of the items on the menu bar and the text that appears on the menu bar for those items.
- Any special attributes for each item.

The **MenuBar** contact is specified in a MENUBAR statement, which consists of the MENUBAR key word, a unique numeric identifier for the **MenuBar**, and a pair curly brackets ({ }) enclosing one or more of the following menu definition statements:

- The MENUITEM statement defines a **MenuItem** contact.
- The MENU statement defines a **Menu** contact which contains a list of menu items.

There are two methods of defining a menu in a resource script. In the first method you must include a full definition of every **MenuItem** contact. The following example defines a **MenuBar** identified by the number 100:

```
MENUBAR = 100 ❶
{
    MENUITEM = 101 { TITLE = 'Exit' } ❷
    MENU = 102 { ❸
        TITLE = 'Format'
        MENUITEM = 103 { TITLE = 'Character' } ❹
        MENUITEM = 104 { TITLE = 'Paragraph' }
    }
}
```

❶ This line tells the resource compiler that this is the start of the definition for a **MenuBar** and assigns the identifier 100 to the new contact. The identifier is used as an handle when you load the contact into your application (see next section). All resource script statements consist of a key word (in this case MENUBAR), an equals sign followed by a numeric identifier, and a pair of curly brackets ({ }) which enclose the item definition statements for that object.

❷ This MENUITEM statement defines the first item on the menu bar. The text 'Exit' will appear as the leftmost command on the menu bar.

❸ The MENU statement defines a pull-down menu, in this case with the title 'Format'. When the user selects the Format command, a menu appears that allows the user to choose between the Character and Paragraph commands.

❹ Within the MENU statement are the definitions for the items on that pull-down menu. The Format menu contains two menu items, each with its own name and title.

When the user selects the Exit, Character or Paragraph commands, UIMS will send the application a **UIMS.MSG.MENUITEM** message that includes the handle of the selected item. Note, however, that UIMS does not notify the application when the user selects the Format command; instead, UIMS displays the Format menu.

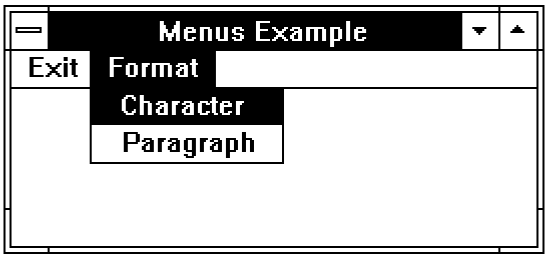


Figure 8-2. Defining a Menu

The second method of defining a menu uses the resource compiler to create the menu item definitions. In this case, the MENUITEM statements within the Format menu definition are replaced by a single CHILDREN statement which lists the titles of the **MenuItem** contacts required. The titles must be enclosed in single quotes. The following example shows how this method can be used to define the same menu as above:

```
MENUBAR = 100
{
    MENUITEM = 101 { TITLE = 'Exit' }
    MENU = 102
    {
        TITLE = 'Format'
        CHILDREN = 'Character' = 103, 'Paragraph' = 104 ❶
    }
}
```

❶ This line lists the menu items that will appear on the Format Menu. When the resource file is compiled, **MenuItem** contacts will be created with the specified titles and identifiers

Note that the second method only allows you to specify some of the possible menu item attributes. You can, for instance, check or disable a menu item, but cannot set the item into auto-check mode.

For more information about the MENUBAR, MENU and MENUITEM resource compiler statements, see the *UIMS DATA/BASIC API Reference Manual*.

Including the Menu in your Application

Once you have defined a menu bar and menus in your resource script, you can include them in your application code. This is done in the same way as for other resources; that is by using the **LoadAppRes** subroutine to fetch handles to the contacts you have defined.

The example below shows how to fetch the handles of the menu items defined in the examples given in the previous section. The menu resources are loaded from the file MENUS.RES.

```
CALL LoadAppRes(CONTEXT, "MENUS.RES", ERR)

IF ERR THEN
    ERROR.STRING = "Failed to load resource file: ":ERR
    GOTO ERROR.EXIT
END
```

Processing Input from a Menu

When a user chooses a command from a menu, UIMS sends a message to the application. The message includes the handles of the **MenuItem** contact and the window to which that contact is attached. The application's message loop should include tests for the window, the message type and the menu item. The example below shows the recommended way of doing this.

```
USER.WANTS.TO.EXIT = FALSE
LOOP UNTIL USER.WANTS.TO.EXIT DO

    CALL GetMsg(0, ...
                MSG.CONTEXT, ...
                MSG.WINDOW❶, ...
                MSG.CONTACT❸, ...
                MSG.TYPE❷, ...
                TS, ...
                DATA1, ...
                DATA2, ...
                DATA3, ...
                VDATA4)

    BEGIN CASE
    CASE WINDOW = WIN1 ❶

        BEGIN CASE

        CASE MSGTYPE = UIMS.MSG.MENUITEM ❷
            GOSUB HANDLE.WIN1.MENU

        REM         case statements for other WIN1 messages

        END CASE

    REM         case statements for other windows

    END CASE

REPEAT
    .
    .
    .
```

```
HANDLE.WIN1.MENU:
    BEGIN CASE

    *       MSG.CONTACT is set up by the call to GetMsg
    CASE MSG.CONTACT = EXIT ❸
        .
        .   REM perform operations for exiting the application
        .

    CASE MSG.CONTACT = FORMAT.CHAR
        .
        .   REM     perform operations for formatting characters
        .

    CASE MSG.CONTACT = FORMAT.PARA
        .
        .   REM     perform operations for formatting paragraphs
        .

    END CASE
RETURN
```

❶ The third parameter to **GetMsg** is a variable (MSG.WINDOW) in which the handle of the window in which the event occurred is returned.

❷ The fifth parameter to **GetMsg** returns the message type.

❸ The fourth parameter to **GetMsg** returns the handle of the contact in which the event occurred.

Modifying Menus from within your Application

UIMS provides functions that you can use to change existing menus and create new menus while your application runs. This section explains:

- How to enable and disable menus and menu items.
- How to check and uncheck menu items.
- How to add, change and delete menu items.
- How to replace a menu.
- How to create and initialise a menu.

Enabling and Disabling Menu Items

Normally the items on a menu are enabled; their text appears normal and the user can select them. Disabled items appear with dimmed (greyed) text and do not respond to mouse clicks or keyboard selection. You should disable a menu item when the command it offers is not appropriate. For example, you might disable the Paste command when there is nothing on the Clipboard.

Setting the Initial State of a Menu Item

You can specify in the resource script that a menu or menu item is disabled. For **MenuItem** contacts this can be done by including a plus sign in the appropriate menu item title when you list the children of a **MenuBar** or **Menu**. For example:

```
MENU = 102
{
    TITLE = 'Edit'
    CHILDREN = 'Cut+' = 103, 'Copy+' = 104, 'Paste' = 105
}
```

defines an Edit menu with Cut, Copy and Paste items. The plus signs following the titles Cut and Copy disable these items. Note that the plus signs are removed from the title text and the **MenuItem** names by the resource compiler.

A pull-down menu can only be defined as disabled by using the ENABLE resource script statement. For example:

```
MENUBAR = 150
{
    MENU = 151
    {
        TITLE = 'Font'
        CHILDREN = 'Times'=160,
                   'Helvetica'=161,
                   'Courier'=162,
                   'Symbol'=163
        ENABLE = FALSE
    }
}
```

defines a menu bar with a Font menu containing Times, Helvetica, Courier and Symbol commands. The line

```
        ENABLE = FALSE
```

disables this menu. Initially, its title will be greyed and it will not appear when the user tries to select it. When it is needed, it can be enabled as described below.

Disabling a Menu Item

A menu item or pull-down menu can be disabled by calling either the Disable subroutine, or the SetEnabled subroutine with the Enabled parameter set to FALSE. For example:

```
CALL Disable(CONTEXT, 105, ERR)
```

and

```
CALL SetEnabled(CONTEXT, 105, FALSE, ERR)
```

both disable the Paste item (identifier 105) on the Edit menu defined in the first example above. ERR is a variable in which a completion code will be returned.

Enabling a Menu Item

A disabled menu item or pull-down menu can be re-enabled by either calling the **Enable** subroutine or specifying the *Enabled* parameter as **TRUE** in a call to **SetEnabled**. For example, the following calls both enable the Cut item (identifier 103) on the Edit menu defined above:

```
CALL Enable(CONTEXT, 103, ERR)

CALL SetEnabled(CONTEXT, 103, TRUE, ERR)
```

Checking and Unchecking Menu Items

A check mark can be displayed next to a menu item to indicate that the user has selected it. You might, for instance, use a check mark to indicate the user's most recent choice from a mutually exclusive group, or to show which options the user has selected from a range of choices. Typically, you would use checked menu items when you want the user to be able to find out easily what is selected.

Setting an Initial Check Mark

You can specify in the resource script whether or not a menu item is checked by including an exclamation mark in the appropriate **MenuItem** title when you list the children of a **MenuBar** or **Menu**. For example:

```
MENU = 120
{
    TITLE = 'View'
    CHILDREN = 'Tools!' = 121, 'Palette' = 122, 'Status Bar!' = 123
}
```

defines a View menu with Tools, Palette and Status Bar items. The exclamation marks following the titles Tools and Status Bar specify that these items are initially to be checked. Note that the exclamation marks are removed from the title text and the **MenuItem** names by the resource compiler.

Checking a Menu Item

A menu item can be checked by calling either the **MenuItemCheck** subroutine, or the **MenuItemSetCheckMark** subroutine with the *Check* parameter set to **TRUE**. For example:

```
CALL MenuItemCheck(CONTEXT, 122, ERR)
```

and

```
CALL MenuItemSetCheckMark(CONTEXT, 122, TRUE, ERR)
```

both set a check mark for the Palette item (identifier 122) on the View menu defined above. ERR is a variable in which a completion code will be returned.

Removing a Menu Item Check Mark

A check mark can be removed from a menu item either by calling the **MenuItemUncheck** subroutine, or by using the **MenuItemSetCheckMark** subroutine with the *Check* parameter set to **FALSE**. For example, both the following examples remove the check mark from the Tools item (identifier 121) on the View menu defined above.

```
CALL MenuItemUncheck(CONTEXT, 121, ERR)

CALL MenuItemSetCheckMark(CONTEXT, 121, FALSE, ERR)
```

Auto-check – Letting UIMS do the Work

If you prefer, you can ask UIMS to do the work of checking and unchecking a menu item, by setting the Auto-check attribute. If this is done, when the user selects the menu item concerned, the checked state will be toggled – if it is checked it will become unchecked and vice versa.

You can set the Auto-check attribute for a menu item in your resource script or when your application is running. The following example shows how this is done in a resource script:

```
MENU = 120
{
   TITLE = 'View'
   MENUITEM = 121
   {
      TITLE = 'Tools'
      AUTOCHECK = TRUE ❶
   }
   .
   .
   .
  }
}
```

❶ This line sets the Auto-check attribute for the Tools item on the View menu.

Note that the Auto-check attribute cannot be set in a list of menu CHILDREN (see page 8-4).

To set the Auto-check attribute while your application is running, use the **MenuItemSetAutoCheck** subroutine. The example below has the same effect as the resource script example above:

```
CALL MenuItemSetAutoCheck(CONTEXT, 121, TRUE, ERR)
```

You can switch off Auto-check by calling **MenuItemSetAutoCheck** with the third parameter set to FALSE.

Changing a Menu

You can change a menu bar or pull-down menu by adding, changing and deleting menu items.

Adding a Menu Item

A new menu item can be inserted before an existing item, or added to the end.

You insert a menu item by using the **AddChild** subroutine. The item you add must have been created with no parent (either in the resource script or while the application is running) and any initialisation (such as disabling, checking, etc.) must have been carried out before calling **AddChild**.

The following example creates a menu item with the title Curve, disables it and then inserts it as the third item on the Draw menu.

```
CALL CreateMenuItem(CONTEXT, 201, "Curve", "", DRAW.CURVE)
CALL Disable(CONTEXT, DRAW.CURVE, ERR)
CALL AddChild(CONTEXT❶, DRAW❷, 2❸, DRAW.CURVE❹, ERR❺)
```

❶ The first parameter to **AddChild** is the handle of the application context.

❷ The second parameter to **AddChild** is the handle of the **Menu** or **MenuBar** in which you are inserting the item; in this case, this is specified by the identifier, DRAW.

❸ The third parameter is the position at which to insert the item. New items are inserted before that at the position you specify. Numbering starts from 0, so the third item is in position 2.

❹ The fourth parameter is the variable containing the handle of the new menu item.

❺ The final parameter is a variable in which a completion code can be returned.

Refer to page 8-15 for details of the **CreateMenuItem** subroutine.

Note: You can add a new pull-down menu to a menu bar in the same way.

If you want to add a new item to the end of a menu, you do not need to know how many items there are already. Simply specify a position of -1 when you call **AddChild**. For example:

```
CALL AddChild(CONTEXT, DRAW, -1, DRAW.RECT, ERR)
```

This adds the Rectangle item at the end of the Draw menu.

Note: If necessary, you can add several items at once by calling **AddChildren** instead of **AddChild**.

Deleting a Menu Item

There are two ways of removing an item from a menu or menu bar.

- You can use **RemoveChild** to remove a specific item. The following example removes the Circle item from the Draw menu:

```
CALL RemoveChild(CONTEXT, DRAW, DRAW.CIRCLE❶, ERR)
```

❶ This parameter is the handle of the item to be removed.

- You can use **RemoveChildren** to remove the item at a specified position. For example, the following call removes the fifth item from the Draw menu:

```
CALL RemoveChildren(CONTEXT, DRAW, 4❶, 1❷, ERR)
```

❶ This parameter specifies the position of the item to be deleted. Remember that numbering starts from 0, so the fifth item is number 4.

❷ This parameter is the number of items to be removed.

In both cases any subsequent items are moved up to fill the gap.

Changing a Menu Item

Items on a menu bar or menu can be enabled and disabled as described on page 8-8 and checked and unchecked as described on page 8-10. If, however, you want to replace one item with another there are two ways of doing this.

- You can remove the old item and replace it with a different one.

```
CALL RemoveChildren(CONTEXT, DRAW, 3, 1, ERR)
CALL AddChildren(CONTEXT, DRAW, 3, DRAW.ARC, ERR)
```

This example removes the fourth item from the Draw menu and then adds an Arc item to replace it.

Note: If you use this method to change items on a menu bar, you will need to prevent the menu bar being re-drawn while the changes are taking place. This can be done by calling the **SetUpdate** subroutine before starting to make your changes and again when you have finished. For example:

```
REM     get the current update status
CALL GetUpdate(CONTEXT, WIN1.MENUBAR, UPDATE❶)

REM     turn off updating
CALL SetUpdate(CONTEXT, WIN1.MENUBAR, UIMS.NONE❷, ERR)

REM     remove the third menu
CALL RemoveChildren(CONTEXT, WIN1.MENUBAR, 2, 1, ERR)
```

```
REM     replace it with the Text menu
CALL AddChildren(CONTEXT, WIN1.MENUBAR, 2, TEXT, ERR)

REM     draw the new menu bar
CALL Draw(CONTEXT, WIN1.MENUBAR, ERR)

REM     restore the previous update status
CALL SetUpdate(CONTEXT, WIN1.MENUBAR, UPDATE, ERR)
```

❶ The third parameter to the **GetUpdate** subroutine is a variable in which to return the current update status.

❷ The third parameter to **SetUpdate** specifies the required update status. A value of **UIMS.NONE** disables updating for the contact.

- You can change the title of an existing menu or menu item by using the **MenuSetTitle** or **MenuItemSetTitle** subroutine as appropriate.

  ```
  CALL MenuItemSetTitle(CONTEXT, CIRCLE, "Ellipse", ERR)
  ```

  This example changes the title of the Circle menu item to Ellipse.

  Note that although the title displayed on the screen for the item has changed, the **MenuItem** contact still has the same handle. Your program will therefore need some way of telling the difference when you process messages relating to this contact in your message loop.

Replacing a MenuBar

A complete menu bar can be removed and replaced with a different one. You would typically do this when you application changes modes and needs a completely new set of commands. For example, you might replace a limited set of commands with a full set when you load a file.

When you replace a window's menu bar, you must use the **AppWinSetMenuBar** subroutine. The example which follows uses **AppWinGetMenuBar** to retrieve the handle of the **MenuBar** currently attached to an **AppWindow**, and saves it for restoring later. It then attaches a new menu bar.

```
CALL AppWinGetMenuBar(CONTEXT, WIN1.HANDLE❶, OLD.WIN1.MENUBAR❷)
CALL AppWinSetMenuBar(CONTEXT, WIN1.HANDLE❶, FULL.MENUBAR❸, ERR)
```

❶ In both subroutines, the first and second parameters are the handle of the application context, and the handle of the window concerned respectively.

❷ **AppWinGetMenuBar** requires a variable in which to return the handle of the menu bar.

❸ The third parameter to **AppWinSetMenuBar** is the handle of the new **MenuBar** contact. The old menu bar is automatically detached from the window and becomes an orphan (it no longer has a parent).

Creating a New Menu

If you prefer, you can create your menus while your application runs instead of defining them in a resource script. The first step is to create a menu bar, using the **CreateMenuBar** subroutine. When a menu bar is first created, it has no menus or menu items on it, so these are created using the **CreatePullDownMenu** and **CreateMenuItem** subroutines.

The following example creates a menu bar consisting of an Exit menu item and a Format menu. The Format menu consists of two menu items, Character and Paragraph. When the menus are complete, the menu bar is attached to the appropriate window.

```
CALL CreateMenuBar(CONTEXT❶, 100❷, ""❸, WIN1.MENUBAR❹)

CALL CreateMenuItem(CONTEXT, 101, "E&xit"❺, WIN1.MENUBAR❸, EXIT)

CALL CreatePullDownMenu(CONTEXT, ...
                        102, ...
                        "&Format"❺, ...
                        WIN1.MENUBAR❸, ...
                        FORMAT)
CALL CreateMenuItem(CONTEXT, "&Character", FORMAT, CHAR)
CALL CreateMenuItem(CONTEXT, "&Paragraph", FORMAT, PARA)
AppWinSetMenuBar(CONTEXT, WIN1, WIN1.MENUBAR, ERR) ❻
```

In this example:

❶ The first parameter to each function is the handle of the application context to which the contact is to belong.

❷ The second parameter to each function is the identifier to be assigned to the contact.

❸ The penultimate parameter is in each case the handle of the contact which is to be the parent of the created contact. In the case of the menu bar this parameter is a null string, so that the contact is created with no parent; the menu bar is only attached to its parent window when the menus are complete. The Exit menu item and the Format menu have the menu bar as their parent, while the Character and Paragraph menu items are attached to the Format menu.

❹ The final parameter to each function is a variable in which the handle of the created contact will be returned.

❺ In the **CreatePullDownMenu** and **CreateMenuItem** calls, the third parameter is the title that will be displayed on the parent menu bar or menu. An ampersand (&) specifies that the next character will act as a selector key that the user can operate to choose the menu or menu item.

❻ This line attaches the menu bar to its parent window.

Note: The **MakePullDownMenu** subroutine provides an alternative method of creating a complete menu, including all its menu items. Refer to the *UIMS DATA/BASIC API, Reference Manual* for details.

Using Cascading Menus

If you need more than one level of pull-down menus, you can create multi-level or cascading menus. Such a menu structure can reduce the number of commands on a single menu, without requiring a dialog box to offer additional choices.

Figure 8-3 shows an example of cascading menus.

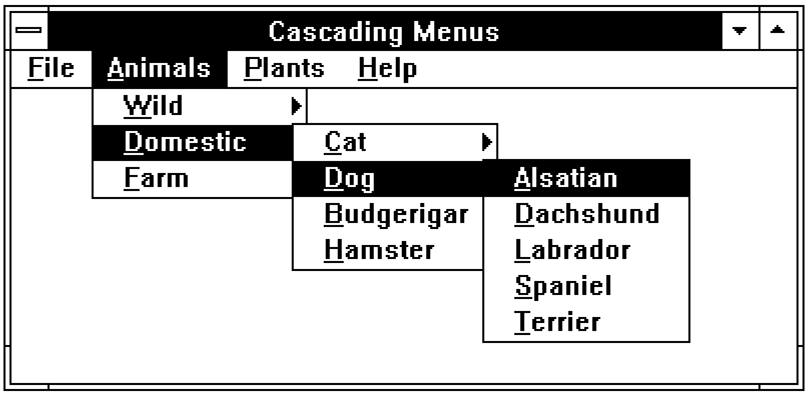


Figure 8-3. Cascading Menus

In this example, the user chose the Domestic command from the Animals menu. At this point the Domestic cascading menu appeared to the right of the cursor. The user then moved the cursor over the Domestic menu and chose the Dog command. The Dog cascading menu then appeared and offered the user the choice of Alsatian, Dachshund, Labrador, Spaniel or Terrier.

Cascading menus are simply nested pull-down menus. The menu definition for the example in Figure 8-3 looks like this:

```
EQUATE idAnimals TO 200
EQUATE idWild TO 210
EQUATE idDomestic TO 220
EQUATE idCat TO 230
EQUATE idDog TO 240
EQUATE idAlsatian TO 241
EQUATE idDachshund TO 242
EQUATE idLabrador TO 243
EQUATE idSpaniel TO 244
EQUATE idTerrier TO 245
EQUATE idBudgie TO 250
EQUATE idHamster TO 260
EQUATE idFarm TO 270

MENU = idAnimals
{
    TITLE = '&Animals'

    MENU = idWild
    {
        TITLE = '&Wild'
        /* definition for Wild menu */
    }

    MENU = idDomestic
    {
        TITLE = '&Domestic'
        MENU = idCat
        {
            TITLE = '&Cat'
            /* definition for Cat menu */
        }

        MENU = idDog
        {
            TITLE = '&Dog'
            CHILDREN = '&Alsatian' = idAlsatian,
                '&Dachshund' = idDachshund,
                '&Labrador' = idLabrador,
                '&Spaniel' = idSpaniel,
                '&Terrier' = idTerrier
        }

        MENUITEM = idBudgie { TITLE = '&Budgerigar'}
        MENUITEM = idHamster { TITLE = '&Hamster'}
    }

    MENU = idFarm
    {
        TITLE = '&Farm'
        /* definition for Farm menu */
    }
}
```

An Example Application: EditMenu

The EditMenu application illustrates the two most common menus: the File menu and the Edit menu. To create the EditMenu application, copy and rename the Generic source files as described in Chapter 4. Then do the following:

1. Add the File and Edit menus to the resource script.
2. Add the menu definitions to the header file.
3. Modify the UIMS.MSG.MENUITEM case in the message loop.
4. Compile the resource file and the DATA/BASIC program.

Although EditMenu has Cut, Copy and Paste menu items, it does not show you how to use the clipboard. This is described in Chapter 11.

Add New Menus to the Resource File

You need to add File and Edit menus to the definition of the App window. Edit the resource script and find the start of the menu bar definition: that is, the lines that contain:

```
MENUBAR = 0
{
```

After the opening brace (curly bracket), add the following:

```
MENU = 0
{
  TITLE = '&File'
  CHILDREN = '&New'         = FileNew,
             '&Open...'     = FileOpen,
             '&Save'        = FileSave,
             'Save &As...' = FileSaveAs,
             '&Print'       = FilePrint,
             '-'            = 0,
             'E&xit'        = FileExit
}

MENU = 0
{
  TITLE = '&Edit'
  CHILDREN = '&Undo+'       = EditUndo,   /* disabled */
             '-'            = 0,
             'Cu&t'         = EditCut,
             '&Copy'       = EditCopy,
             '&Paste+'     = EditPaste,  /* disabled */
             'C&lear'      = EditClear
}
```

The File menu has six commands and a separator; each command has a mnemonic (selector key), indicated by the ampersand (&). Similarly, the Edit menu has five commands and a separator, each command having a mnemonic.

The separators between the Print and Exit commands on the File menus, and the Undo and Cut commands on the Edit menu, place horizontal bars between these commands on the menu. A separator is recommended between menu commands that otherwise have nothing in common. For example, Undo affects only the application, whereas the remaining commands affect the clipboard.

Add Definitions to the Header File

Each menu item handle (identifier) must be declared in your application's header file, because these constants are used in both the resource script and the DATA/BASIC source.

Edit your header file and find the line:

```
EQUATE HelpAbout TO 20
```

Replace this line with the following:

```
* File menu

EQUATE FileNew     TO 101
EQUATE FileOpen    TO 102
EQUATE FileSave    TO 103
EQUATE FileSaveAs  TO 104
EQUATE FilePrint   TO 105
EQUATE FileExit    TO 106


* Edit menu

EQUATE EditUndo    TO 111
EQUATE EditCut     TO 112
EQUATE EditCopy    TO 113
EQUATE EditPaste   TO 114
EQUATE EditClear   TO 115
```

```
* Help menu

EQUATE HelpAbout   TO 121
```

Modify the Menu Item Case

The EditMenu application is intended to demonstrate how to process menu commands. Therefore, instead of performing tasks, most of the menu commands display the message "Command not implemented!". The exceptions are the Help About command, which displays an About box, and the File Exit command, which closes the application.

Edit your DATA/BASIC source file and, in the HANDLE.WIN1.MENUS subroutine, find the line which contains:

```
CASE MSG.CONTACT = HelpAbout
```

Just before this line, insert the following:

```
  * File menu commands
  CASE MSG.CONTACT = FileNew OR ...
       MSG.CONTACT = FileOpen OR ...
       MSG.CONTACT = FileSave OR ...
       MSG.CONTACT = FileSaveAs OR ...
       MSG.CONTACT = FilePrint

    CALL CreateMessageBox(CONTEXT, ...
                          UIMS.INFO, ..
                          "EditMenu Example Application", ...
                          "Command not implemented!", ...
                          "", ...
                          OK, ...
                          ERR)

  CASE MSG.CONTACT = FileExit
    USER.WANTS.TO.EXIT = TRUE

  * Edit menu commands
  CASE MSG.CONTACT = EditUndo OR ...
       MSG.CONTACT = EditCut OR ...
       MSG.CONTACT = EditCopy OR ...
       MSG.CONTACT = EditPaste OR ...
       MSG.CONTACT = EditClear

    CALL CreateMessageBox(CONTEXT, ...
                          UIMS.INFO, ...
```

```
"EditMenu Example Application", ...
"Command not implemented!", ...
"", ...
OK, ...
ERR)
```

Compile

When you have made these changes, you can compile the resource script and the DATA/BASIC program as described in Chapter 4 for the Generic application. When you run the application, you will notice the following:

- The Undo and Paste items on the Edit menu are displayed in grey rather than black, indicating that they are disabled.
- The About command on the Help menu displays an About box.
- The Exit command on the File menu closes the application.
- The remaining commands display the message "Command not implemented!".

Chapter 9
Controls

Controls are special windows that provide various ways of interacting with the user.

This chapter covers the following topics:

- What is a control?
- Creating a control.
- Using a control.
- Using the different types of control in application and child windows.

An example application which illustrates these concepts is also described.

What is a Control?

A control is a pre-defined child window that carries out a specific kind of input or output. For example, to request a filename from the user, you can create and display an edit control in which the user must type the name. An edit control is a pre-defined child window that receives and displays keyboard input.

UIMS provides controls which allow the user to input text, select options and values and initiate operations. You can also use controls to give the user information.

The following controls are available:

- Titled, option and check button controls.
- Static controls – text, line and rectangle.
- List boxes.
- Edit controls – edit box and text editor.
- Scroll bars.
- Inclusive and exclusive groups.

Creating a Control

You create a control in the same way as other contacts: either by defining it in your resource script, or by calling the appropriate create subroutine. The parameters required depend on the control, but in general you must supply the following:

- The handle of the application context.
- A number by which the contact can be identified.
- The handle of the parent object. Alternatively you can create the control without a parent and attach it at a later time.
- The position of the object on the screen relative to its parent.
- The size of the object. This can be calculated automatically, if required.
- A variable in which to return the handle of the created control.
- Control-specific information, such as the title of a button or whether or not to display scroll bars in a text editor.

Both methods of creating controls are illustrated later in this chapter.

Setting the Parent Window

As mentioned above, you can specify a parent object when you create a control. Note, however, that if the parent is displayed on the screen at the time the new contact will be displayed immediately and in some cases this may not be desirable – if, for instance, you need to set additional attributes before using the control. Under these circumstances you should create the contact without a parent and attach it using **AddChild** or **AddChildren** once you have completed your initialisation.

As with any child window, changes to the parent object also affect the control. For example, if you disable the parent , the control will be disabled as well. If UIMS paints, moves or destroys the parent, it also paints, moves or destroys the control.

Although a control can be any size and can be moved to any position, it is restricted to the client area of its parent. UIMS clips the control if you move it outside the parent's window area, or make it bigger than the client area.

Using a Control

Once you have created a control, you can:

- Receive user input through the control.
- Tell the control to perform specialised tasks, such as returning a string of text.
- Enable or disable input to the control.
- Move the control or change its size.
- Destroy the control.

This section describes how to perform these tasks.

Receiving User Input

As the user interacts with a control, it sends a message containing information about that interaction to the application. The type and content of the message will depend on the control and the operation performed, but it will always include the handles of the contact and the parent window. For example, when the user clicks on a button control, a **UIMS.MSG.BUTTONPRESS** message is generated.

Control Tasks

Each control has a number of related subroutines which can be used to change its attributes and to return its current attribute settings. For example, the **EditBoxGetContent** subroutine returns the text displayed in an edit box control.

In addition there are subroutines connected with a number of attributes that are common to almost all controls. For example, you can find out the size of a control by calling the **GetSize** subroutine.

Disabling and Enabling Input to a Control

If a particular control is not appropriate in the current state of your application you can disable it. Disabled items are shown dimly (are greyed) and do not respond to mouse clicks or keyboard selection.

To disable a control, call the **SetEnabled** subroutine and specify **FALSE** as the required state. For example:

```
CALL SetEnabled(Context.Handle, Control.Handle, FALSE, ERR)
```

To re-enable a disabled control, simply call **SetEnabled** with the second parameter set to **TRUE**:

```
CALL SetEnabled(Context.Handle, Control.Handle, TRUE, ERR)
```

Moving and Sizing a Control

To move a control within its parent, call the **Move** subroutine. This positions the top left-hand corner of the control relative to the top left-hand corner of its parent's client area. The following example illustrates this.

```
CALL Move(Context.Handle, Control.Handle, 10, 40, ERR)
```

This moves the specified contact to a position 10 coordinate units to the left of, and 40 coordinate units down from the top left-hand corner of its parent's client area.

Note that UIMS automatically moves a control when it moves the parent window.

If you want to change the size of a control, call the **Resize** subroutine. The following example makes a contact 30 coordinate units wide by 12 units high.

```
CALL Resize(Context.Handle, Control.Handle, 30, 12, ERR)
```

Destroying a Control

You can destroy a control by calling the **Destroy** subroutine. This deletes any internal record of the control and removes it from its parent's client area. The following example shows how this is done:

```
CALL Destroy(Context.Handle, Control.Handle, ERR)
```

Note that UIMS automatically destroys a control when its parent is destroyed.

Button Controls

A button control is a small window used for input of the simple yes/no, on/off type. The following button controls are available:

- Titled Button.
- Check Button.
- Option Buttons and Exclusive Groups.

Titled Buttons

A titled button is a button that the user selects to carry out an action; the button contains text or a graphic that indicates what it does. When the user clicks a titled button, the application normally carries out the associated action immediately. For example, if the user clicks the Cancel button in a dialog box, the application immediately removes the dialog box and cancels any changes the user may have made to the dialog.

As with other contacts, you can create a titled button either in your resource script or while your application is running. The following examples both create a **TitledButton** contact with the title 'Cancel'.

The first example shows how this is done in the resource script:

```
EQUATE Win1 TO 100
EQUATE CancelButton TO 101

APPWINDOW = Win1
{
    .
    . /* definition for Win1 */
    .

    TITLEDBUTTON = CancelButton
    {
        TITLE = 'Cancel'
        POSITION = 80, 24
        SIZE = 60, 18
    }
    .
    . /* further definitions for Win1 */
    .
}
```

The three **TitledButton** attributes shown above (TITLE, POSITION and SIZE) must be specified. If any of these is missing, your resource script will not compile.

Note: You cannot create a titled button containing a graphic in your resource script.

Within your application, you use the **CreateTitledButton** subroutine:

```
CALL CreateTitledButton(CONTEXT❶, ...
                        101❷, ...
                        "c:\rfw\vcrend.bmp"❸, ...
                        80, 24❹, ...
                        60, 18❺, ...
                        WIN1❻, ...
                        VCREND❼)
```

In this example:

❶ The first parameter is the handle of the application context.

❷ The second parameter is a number by which the button can be identified.

❸ The third parameter is the title that will be displayed on the button or, in this case, the name of a graphics file.

❹ The next two parameters specify the position of the button relative to the top left-hand corner of its parent window. These values will be interpreted as character or pixel units, depending on the current coordinate mode of the application context. In this case the top left corner of the button will be 80 units to the right of and 24 units down from the top left corner of the parent window.

❺ The sixth and seventh parameters specify the size of the button. As in the case of the position parameters, these values will be interpreted as character or pixel units, depending on the current coordinate mode of the application context. In this case the button will be 60 units wide and 18 units high. Note that if these parameters are both zero, the size of the button will be calculated automatically.

❻ The eighth parameter is the handle of the parent window. If you wish to change any of the button's default attributes before displaying it, you can create the button without a parent by passing a null string instead of a handle. You can then make the changes you require before attaching the button to its parent with **AddChild** or **AddChildren**.

❼ The final parameter is a variable in which to return the handle to the newly created button.

Note that because the button created in this example does not form part of a dialog box, the application must carry out the actions appropriate to the button, such as (in this case) removing the button from the screen and discarding changes.

A titled button responds to mouse and keyboard input by sending a message to the application. Mouse clicks and SPACEBAR operations generate **UIMS.MSG.BUTTONPRESS** messages and all other key operations generate **UIMS.MSG.KEYPRESS** messages.

Default Titled Buttons

A default titled button is normally used so that the user can signal the completion of some activity without having to specifically select the button concerned. An typical example is in a dialog box that asks the user for a file name. When the file name has been typed into an edit control, the user can simply press RETURN to complete the action, instead of having to specifically select the default, OK button.

The default titled button is normally indicated to the user by a thickened border, which is specified by including the style **THICK** in the contact definition. The following resource script example creates an OK button with a thickened border:

```
TITLEDBUTTON = 115
{
    TITLE = 'OK'
    POSITION = 240, 80
    SIZE = 60, 18
    STYLE = THICK
}
```

If you create a titled button while your application is running, you cannot specify the style with **CreateTitledButton**. Instead, you must change the style after the button has been created by using the **TitledButtonSetStyle** subroutine. The example below creates an OK button with no parent, gives it a thickened border and then attaches it to a window:

```
CALL CreateTitledButton(CONTEXT, 115, "OK", 240, 80, 60, 18, "", ...
                          OK.BUTTON)
CALL TitledButtonSetStyle(CONTEXT, OK.BUTTON, UIMS.TB.THICK, ERR)
CALL AddChild(CONTEXT, WIN1, -1, OK.BUTTON, ERR)  ❶
```

❶ This line attaches the OK button to a child of the window WIN1. The second parameter (-1) specifies that the button is to be added to the end of the list of children for WIN1.

Note that the final parameter to both **TitledButtonSetStyle** and **AddChild** is a variable in which to return a completion code. In a practical application, this variable would be tested to detect any errors.

Check Buttons

A check button typically allows the user to select an option to use in the current task. By convention, within a group of check buttons, the user can select more than one option. To present options that are mutually exclusive, use option buttons (see page 9-10).

For example, you might present a group of check buttons to let the user choose the font style for the selected text. The user could choose both bold and italic by choosing both the 'Bold' and 'Italic' check buttons.

The following resource script example creates an Italic check button.

```
CHECKBUTTON = 217
{
    TITLE = 'Italic'
    POSITION = 80, 64
    SIZE = 67, 16
}
```

Note that the TITLE, POSITION and SIZE attributes must be specified.

Within your application, you create a check button with the **CreateCheckButton** subroutine:

```
CALL CreateCheckButton(CONTEXT, 217, "Italic", 80, 64, 0❶, 0❶, ...
                            WINDOW, ITALIC.CHECK)
```

❶ These two parameters respectively specify the width and height of the check button. If both are zero (as in this case), size of the check button will be calculated automatically.

A check button responds to mouse and keyboard input in the same way as a titled button. A **UIMS.MSG.BUTTONPRESS** message is generated when the user clicks the control or presses the SPACEBAR. However, an additional function of a check button is to display a check (an X) in its box to show that the option it represents is currently selected.

To display a check in a check button, use the **CheckButtonSetSelected** subroutine. Similarly, to find out whether a check button has a check displayed, use **CheckButtonGetSelected**. The following example places a check in a check button:

```
CALL CheckButtonSetSelected(CONTEXT, CHECK.BUTTON❶, TRUE❷, ERR❸)
```

In this example:

❶ The first parameter is the handle of the check button.

❷ The second parameter must be **TRUE** to display a check, or **FALSE** to remove it.

❸ The final parameter is a variable in which to return a completion code.

If you prefer, you can ask UIMS to do some of the work by enabling auto-toggle for the check button. The button will then automatically display or remove its check (as appropriate) whenever it is actioned. Auto-toggle can be selected in the resource script or by using the **CheckButtonSetToggle** subroutine.

Option Buttons

Option buttons work in a similar way to check buttons, but are usually grouped to represent mutually exclusive options. For example, you might use a group of option buttons to let the user specify the unit of measure used by a publishing program (inches, centimetres, points or picas). The option buttons would allow the user to select only one type of unit at a time.

You create an option button in the same way as you would any other button, either in your resource script or by using the **CreateOptionButton** subroutine. The examples that follow both create an **OptionButton** contact with the title 'Inches'.

In the resource script:

```
OPTIONBUTTON = 284
{
    TITLE = 'Inches'
    POSITION = 306, 120
    SIZE = 80, 16
}
```

The TITLE, POSITION and SIZE attributes must be specified.

While your application is running:

```
CALL CreateOptionButton(CONTEXT, 284, "Inches", 306, 120, 0❶, ...
                                           0❶, WINDOW, INCHES.OPT)
```

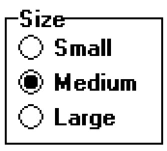
❶ These two parameters respectively specify the width and height of the option button. If both are zero (as in this case), size of the option button will be calculated automatically.

As in the case of check buttons, option buttons can be 'checked' (a solid circle is displayed instead of an X) by calling the appropriate function (**OptionButtonSetSelected** in this case) and can be set to auto-toggle by calling **OptionButtonSetToggle**. Note that, since option buttons are mutually exclusive, when you check an option button you should also clear the check on the previously selected button (if any). You can determine which option button in a group is checked by calling **OptionButtonGetSelected** for each button.

Exclusive Groups

If you wish, you can make your option buttons into an exclusive group. The advantage of this is that UIMS will do the work of clearing the check on the previously selected button when the user makes a new selection. You can also enclose the group in a box and give it a title.

You create an exclusive group in your resource script or with the **CreateExGroup** subroutine and attach the appropriate option buttons as its children. The following examples create an exclusive group with the title Size, containing option buttons entitled Small, Medium and Large. The group is enclosed in a box.



In the resource script:

```
EQUATE ExgrpSize TO 350
EQUATE ObutnSizeSmall TO 351
EQUATE ObutnSizeMedium TO 352
EQUATE ObutnSizeLarge TO 353

EXCLUSIVEGROUP = ExgrpSize
{
    TITLE = 'Size'
    POSITION = 246, 13
    SIZE = 110, 65 ❶
    STYLE = BORDER ❷

    OPTIONBUTTON = ObutnSizeSmall
    {
        TITLE = 'Small'
        POSITION = 10, 12 ❸
        SIZE = 69, 16
    }

    OPTIONBUTTON = ObutnSizeMedium
    {
        TITLE = 'Medium'
        POSITION = 10, 29
        SIZE = 87, 16
        SELECTED = TRUE ❹
    }

    OPTIONBUTTON = ObutnSizeLarge
    {
        TITLE = 'Large'
        POSITION = 10, 46
        SIZE = 72, 16
    }
}
```

❶ The exclusive group must be large enough to contain all the option buttons, even if its border is not displayed.

❷ This line specifies the style of the box enclosing the group. The **BORDER** style gives a single box with square corners.

❸ The positions of the option buttons must be given relative to the origin of the exclusive group. Note that the origin is aligned with the top of the title text and the left-hand edge of the box.

❹ This line selects the Medium option button as the initial selection.

Note that there is no need to set auto-toggle mode for the option buttons. Within an exclusive group, option buttons are always selected automatically when actioned.

To achieve the same while your application is running:

```
CALL CreateExGroup(CONTEXT, 350, "Size", 246, 13, 110, 65, ...
                     UIMS.BORDER❶, "", SIZE)

CALL CreateOptionButton(CONTEXT, 351, "Small", 10, 12, SIZE, ...
                          SIZE.SMALL)
CALL CreateOptionButton(CONTEXT, 352, "Medium", 10, 29, SIZE, ...
                          SIZE.MEDIUM)
CALL CreateOptionButton(CONTEXT, 353, "Large", 10, 46, SIZE, ...
                          SIZE.LARGE)

CALL OptionButtonSelect(CONTEXT, SIZE.MEDIUM, ERR) ❷

CALL AddChild(CONTEXT, WINDOW, -1, SIZE, ERR)
```

❶ This parameter specifies the style of the exclusive group. The **UIMS.BORDER** style encloses the group in a box.

❷ Note the use of the **OptionButtonSelect** subroutine as an alternative to **OptionButtonSetSelected**.

You can find out which button in an exclusive group is selected by calling the **ExGroupGetSel** subroutine.

```
CALL ExGroupGetSel(CONTEXT, 350, SELECTION❶)
```

❶ The handle of the selected option button is returned in this variable.

Static Controls

A static control is a small window that contains text or graphics. These are typically used to label other controls or to create boxes and lines to separate one group of controls from another.

Static controls do not respond to user input; that is, they do not generate UIMS events when selected. However, you can change the appearance and location of a static control at any time by calling the appropriate subroutine (refer to the *UIMS DATA/BASIC API, Reference Manual*). For example, you can change the text associated with a **Text** contact by calling the **TextSetContent** subroutine.

There are three types of static control: the **Text**, **Line** and **Rectangle** contacts.

Text

As with other controls, you can create a **Text** contact either in your resource script or by using the **CreateText** subroutine. In both cases you must specify the text string to be displayed and the position and size of the containing window. Unless changed, the contact will use the same drawrule as its parent. The amount of text displayed is limited by the size of the containing window; text that will not fit into this window is clipped.

When first created, the text is left-aligned relative to its containing window. If required, it can be changed to right aligned, centred or justified (both left and right aligned), by calling the **TextSetJustification** subroutine.

The following example creates a Text contact without a parent, sets it to right-aligned and then attaches it to a window:

```
TEXT = "This text is right-aligned" ❶
CALL CreateText(CONTEXT, 324, TEXT❶, 198❷, 53❷, 100❸, 0❸, "", ...
                RTEXT)
CALL TextSetJustification(CONTEXT, RTEXT, UIMS.JUST.RIGHT❹, ERR)

CALL AddChild(CONTEXT, WIN1, -1, RTEXT, ERR)
```

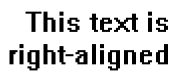
❶ The third parameter to **CreateText** defines the text to be displayed.

❷ These two parameters specify the position of the text within the window.

❸ These two parameters specify the size (width, height) of the text contact. In this case, the height is zero – UIMS will therefore make the text window tall enough to display all the text within the specified width.

❹ The style **UIMS.JUST.RIGHT** specifies right-alignment.

When displayed, the text will appear similar to the following:



This text is
right-aligned

Lines and Rectangles

Lines and rectangles are created in a similar way to static text. For example, the following creates a rectangle with a diagonal line from bottom left to top right:

```
CALL CreateRect(CONTEXT, ...
                257❶, ...
                80, 80❷, ...
                240, 240❸, ...
                0❹, ...
                WIN1, ...
                RECT)
CALL CreateLine(CONTEXT, ...
                258❶, ...
                80, 240❺, ...
                240, 80❻, ...
                0❹, ...
                WIN1, ...
                DIAG)
```

❶ In both cases, the second parameter to the create subroutine is a numeric identifier.

❷ The top left-hand corner of the rectangle.

❸ The size of the rectangle (width, height).

❹ In both cases, the seventh parameter to the create subroutine is provided for future use. It must be present, but its value will be ignored.

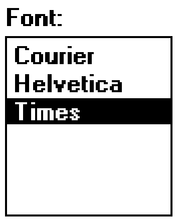
❺ The position of the start of the line (coincides with the bottom left-hand corner of the rectangle).

❻ The position of the end of the line relative to its start. A line can be thought of as the diagonal of an imaginary box; with these parameters specifying the size (width and height) of that box.

List Boxes

A list box is a box that contains a list of selectable items, such as filenames. You would typically use a list box to display a list of items from which the user can select one or more, but where the items available might change. You can also scroll a list box to see items that will not fit into the box, so a list box is useful where you have a large number of choices. If you have a short list which is always the same, you would probably use option buttons or check buttons instead.

You can create a list box in your resource script, or by calling the **CreateListBox** subroutine. If you use **CreateListBox**, the contact is created without any contents; this must be added using the **ListBoxAddContents** subroutine. The following examples both create a list box containing a list of text fonts.



In the resource script:

```
EQUATE FontTitle TO 51
EQUATE FontList TO 52

TEXT = FontTitle ❶
{
    POSITION = 24, 64
    SIZE = 40, 13
    CONTENT = 'Font:'
}

LISTBOX = FontList
{
    POSITION = 24, 77
    SIZE = 110, 96
    CONTROLS = NONE
    CONTENT = 'Courier','Helvetica','Times'
}
```

❶ If you want your list box to have a title, you must use a **Text** contact.

While your application is running:

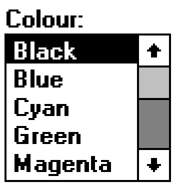
```
EQUATE AM TO CHAR(254)
EQUATE FontTitle TO 51
EQUATE FontList TO 52
    .
    .
    .
CALL CreateText(CONTEXT, FontTitle, "Font:", 24, 64, 40, 13, ...
                  "", FONT.TITLE)
CALL CreateListBox(CONTEXT, FontList, 24, 64, 110, 96, ...
                     UIMS.NONE❶, "", FONT.LIST)

FONTS = "Courier":AM:"Helvetica":AM:"Times" ❷
CALL ListBoxAddContents(CONTEXT, FONT.LIST, 0❸, FONTS, ERR)

CALL AddChild(CONTEXT, WIN1, -1, FONT.TITLE, ERR)
CALL AddChild(CONTEXT, WIN1, -1, FONT.LIST, ERR)
```

❶ In both examples, this parameter defines the type of list box. **NONE** (in the resource script) and **UIMS.NONE** (in your application) both specify a standard list box which allows only one item to be selected at a time.

❷ The items to be displayed in the list box must be in the form of a dynamic array, with one item in each attribute.

❸ This parameter indicates where to add the new items relative to any existing list. In this case there is no existing list, so a position of 0 is used.

In this case the list box is large enough to display all the items at the same time. If, however, there are too many items to show at once, the list box will include a scroll bar. This appears automatically when needed and you should allow for this when setting the width of the contact. For example:



Changing the Contents of a List Box

As can be seen from the example above, you add items to a list box by calling the **ListBoxAddContents** subroutine. A second subroutine, **ListBoxAddContent**, allows you to add a single item to the list box. In both cases, the new items are added before the specified existing item, the first item in the list being numbered 0. Specifying a position of -1 adds the new items to the end of the list.

Note that the **ListBox** contact cannot sort the list. If you want the items presented in alphabetical or any other logical order, you must sort them before calling **ListBoxAddContents**. If you want to add items to a sorted list, you will need to ensure that new items are added in the correct position. One way of doing this is to fetch the contents of the list box, add the new items to the array and then sort the modified list. The old list can then be removed and replaced by the new one. The following example illustrates this:

```
REM fetch the current list box contents
CALL ListBoxGetContents(CONTEXT, FONT.LIST, FONTS➊, ERR)

REM add the new items to the end of the list
FONTS<-1> = "Symbol"
FONTS<-1> = "Palatino"

    .
    .  REM routine to sort the list
    .

REM remove the old list
CALL ListBoxRemoveContents(CONTEXT, FONT.LIST, 0➋, -1➋, ERR)

REM replace it with the revised list
CALL ListBoxAddContents(CONTEXT, FONT.LIST, 0, FONTS, ERR)
```

➊ This is a variable in which to return the list of items. On return this will contain a dynamic array with one item in each attribute.

➋ These two parameters respectively specify the position of the first item and the number of items to be removed. The values shown will remove the complete list.

Note that you can also remove named items by calling the **ListBoxRemoveContent** subroutine (not to be confused with the **ListBoxRemoveContents** routine used above).

Using Standard List Boxes

The standard list box allows the selection of only one item at a time. If the user clicks an item or presses the SPACEBAR in the list box, the list box selects the item concerned (removing the selection from the previously selected item, if any) and indicates the selection by inverting the item text. At the same time, **UIMS.MSG.LBOX.DESELECT** and **UIMS.MSG.LBOX.SELECT** messages are generated, giving the positions in the list of the items that have changed.

Multiple-selection List Boxes

If you want to allow the user to make multiple selections in a list box, you must change the style of the list box, either by setting the CONTROLS attribute to **MULTISELECT** in your resource script, or by specifying this style when you call the **CreateListBox** subroutine. The examples which follow illustrate these two methods.

In your resource script:

```
EQUATE FruitList TO 78

LISTBOX = FruitList
{
    POSITION = 56, 40
    SIZE = 96, 48
    CONTROLS = MULTISELECT
    CONTENT = 'Apple','Lemon','Lime','Orange','Strawberry'
}
```

While your application is running:

```
EQUATE AM TO CHAR(254)
EQUATE FruitList TO 78
    .
    .
    .
CONTROLS = UIMS.LBOX.MULTISELECT❶
CALL CreateListBox(CONTEXT, FruitList, 56, 40, 96, 48, ...
                     CONTROLS❶, "", FRUIT.LIST)

FRUIT = "Apple":AM:"Lemon":AM:"Lime":AM:"Orange":AM:"Strawberry"
CALL ListBoxAddContents(CONTEXT, FRUIT.LIST, 0, FRUIT, ERR)

CALL AddChild(CONTEXT, WIN1, -1, FRUIT.LIST, ERR)
```

❶ This parameter specifies a multiple-selection list box.

A multiple-selection list box is essentially the same as a standard list box, except that the user can select more than one item at a time. As each new item is clicked, it is added to those already selected.

Edit Controls

An edit control is a rectangular window in which the user can enter and edit text. There are two types of edit control: a single-line **EditBox** contact, and a multi-line **TextEditor**.

Edit Box

The **EditBox** contact allows the entry of a single line of text. Various styles and attributes are available for the contact, set when the edit box is created or by calling different subroutines (refer to the *UIMS DATA/BASIC API, Reference Manual* for full details).

The following example shows how to define an edit box in your resource script:

```
EQUATE EBox = 72

EDITBOX = EBox
{
    POSITION = 136, 80
    SIZE = 72, 18
    STYLE = BORDER ❶
    MASK = '' ❷
}
```

❶ You must specify a style for the edit box. **BORDER** encloses the contact in a rectangular box.

❷ The MASK attribute is for use in the future – you must specify a null string

The same contact can be created from within your application by calling the **CreateEditBox** subroutine:

```
CALL CreateEditBox(CONTEXT, ...
                   72❶, ...
                   136, 80, ...
                   72, 18, ...
                   UIMS.EBOX.BORDER❷, ...
                   ""❸, ...
                   ""❹, ...
                   EBOX)
```

❶ This parameter is the numeric identifier for the new edit box.

❷ This parameter specifies the style of the edit box. **UIMS.EBOX.BORDER** encloses the contact in a rectangular box.

❸ This parameter (Mask attribute) is for use in the future – you must specify a null string.

❹ This parameter specifies the parent of the edit box. A null string specifies that the contact should have no parent.

The contents of an edit box can be obtained by calling **EditBoxGetContent**. If it is necessary to set the contents to a default value, this can be done with **EditBoxSetContent**.

A typical use for an edit box is in combination with a list box, so that the user can select an item from the list box for editing. A link can be set up between the list box and the edit box by calling the **ListBoxSetLink** subroutine; a selection made in the list box will then be automatically copied into the edit box.

Text Editor

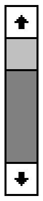
If you want to allow the user to type more than one line of text, you must use the **TextEditor** contact. The contact has various attributes which can be set when the text editor is created or by calling different subroutines (refer to the *UIMS DATA/BASIC API, Reference Manual* for full details).

When the user selects text within a **TextEditor** contact, a message sent to the application. This message includes details of the line number, and the starting and finishing character positions within that line.

Scroll Bars

Scroll bars are most frequently seen in association with other contacts: application windows, child windows and list boxes for instance. They can, however, also be used as independent controls positioned anywhere in a parent window. They allow the user to select a value from an unbroken range of values. Whenever the user clicks within the scroll bar with the mouse, or the moves the thumb with the keyboard, a message is sent to the application. The application can use this message to determine the value selected by the user and to carry out the appropriate actions.

A scroll bar is created by calling the **CreateScrollBar** subroutine, specifying whether you require a vertical or a horizontal scroll bar. If you are using graphics coordinate mode you can set the width (of a vertical scroll bar) or height (of a horizontal scroll bar) to match the size of a window scroll bar by calling **DisplayGetMetrics** to fetch the required value. The following example creates a vertical scroll bar the same width as a standard window scroll bar:



```
CALL GetDefaults(SCRN, PRINTER, TYPEFACE, ERR) ❶
CALL DisplayGetMetrics(CONTEXT, SCRN, P3, P4, P5, P6, P7, P8, ...
                         VSWIDTH, HSHEIGHT, ERR) ❷

WIN1.SCRL = 72
STYLE = UIMS.SCROLLBAR.VERT
CALL CreateScrollBar(CONTEXT, WIN1.SCRL, STYLE, 25, 20, VSWIDTH, ...
                       84, WIN1, WIN1.SCRL) ❸
```

❶ This line fetches the handles of the default screen, printer and typeface. In this case we are only interested in the screen handle.

❷ **DisplayGetMetrics** returns the sizes of various window elements. In this case we are only interested in the width of a vertical scroll bar, which is returned in the ninth parameter (VSCRLWIDTH). If we wanted the height of a horizontal scroll bar, we would use the value returned in the tenth parameter. Parameters P3 to P8 are variables which return the sizes of other elements – refer to the *UIMS DATA/BASIC API, Reference Manual* for full details.

❸ This line creates a standard width vertical scroll bar 84 pixels high and positions it 25 pixels across and 20 pixels down within window WIN1.

When a scroll bar control is operated **UIMS.MSG.SCROLL** messages are generated. For these messages, on return from **GetMsg** the *vData2* variable will contain details of the type of movement and *vData4* the new thumb position. UIMS automatically repositions the thumb, but your application must carry out any other actions. Scroll bars can operate in tracking and non-tracking modes (set by calling **ScrollBarSetTracking**). When tracking is enabled, the thumb position is continually reported as it is dragged; if it is disabled the position is reported only when the thumb is released.

When a scroll bar is created, the minimum and maximum values that it represents are both 0; to establish the required range of values, you must call the **ScrollBarSetRange** subroutine. Similarly, the line and page increments are initially set to 0 and must be set to appropriate values with **ScrollBarSetInc**. For example, if your application has a scroll bar with which the user can select an hour in the day, you would call **ScrollBarSetRange** to set the range to 24 hours, and **ScrollBarSetInc** to set a line increment of 1:

```
CALL ScrollBarSetRange(CONTEXT, SCROLLBAR, 0, 23, ERR)
CALL ScrollBarSetInc(CONTEXT, SCROLLBAR, 3, 1, ERR)
```

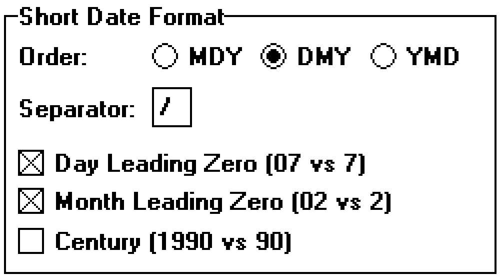
In this example, **ScrollBarSetRange** sets the minimum value to 0 and the maximum to 23, and **ScrollBarSetInc** a page increment of 3 and a line increment of 1.

Note that while UIMS automatically positions the thumb as it is moved with the mouse or the keyboard, if the range is changed your application must reposition the thumb to reflect the new values. Also, if your application allows the user to change the value represented by the thumb position without using the scroll bar (by typing in an edit control, for instance), your application must reposition the thumb as necessary.

Inclusive Groups

Inclusive groups are similar to exclusive groups in that they provide a way of grouping related controls. Unlike exclusive groups, however, they allow the user to make multiple selections and can include several different types of control in the same group.

An inclusive group is created in much the same way as any other control; you can define it in your resource script or use **CreateIncGroup** within your application. The following examples create an inclusive group containing controls that allow the user to set a short format for displaying dates:



Short Date Format

Order: ○ MDY ● DMY ○ YMD

Separator: /

☒ Day Leading Zero (07 vs 7)

☒ Month Leading Zero (02 vs 2)

☐ Century (1990 vs 90)

In the resource script:

```
EQUATE SDate TO 320
EQUATE SDate_Order TO 321
EQUATE SDate_MDY TO 322
EQUATE SDate_DMY TO 323
EQUATE SDate_YMD TO 324
EQUATE SDate_Sep TO 325
EQUATE SDate_Day TO 326
EQUATE SDate_Month TO 327
EQUATE SDate_Cent TO 328

INCLUSIVEGROUP = SDate
{
    TITLE = 'Short Date Format'
    POSITION = 20, 14
    SIZE = 328, 125
    STYLE = BORDER
```

```
TEXT = 0
{
    CONTENT = 'Order:'
    POSITION = 9, 18
    SIZE = 51, 12
}

EXCLUSIVEGROUP = SDate_Order
{
    TITLE = ''
    POSITION = 93, 10
    SIZE = 220, 30
    STYLE = NONE

    OPTIONBUTTON = SDate_MDY
    {
        TITLE = 'MDY'
        POSITION = 5, 6
        SIZE = 65, 16
    }

    OPTIONBUTTON = SDate_DMY
    {
        TITLE = 'DMY'
        POSITION = 78, 6
        SIZE = 65, 16
        SELECTED = TRUE
    }

    OPTIONBUTTON = SDate_YMD
    {
        TITLE = 'YMD'
        POSITION = 151, 6
        SIZE = 65, 16
    }
}

TEXT = 0
{
    CONTENT = 'Separator:'
    POSITION = 9, 42
    SIZE = 84, 12
}
```

```
    EDITBOX = SDate_Sep
    {
        POSITION = 98, 39
        SIZE = 27, 18
        STYLE = BORDER
        JUSTIFICATION = LJUST
        MASK = '1?'
    }

    CHECKBUTTON = SDate_Day
    {
        TITLE = 'Day Leading Zero (07 vs 7)'
        POSITION = 9, 65
        SIZE = 236, 16
        TOGGLE = TRUE
        SELECTED = TRUE
    }

    CHECKBUTTON = SDate_Month
    {
        TITLE = 'Month Leading Zero (02 vs 2)'
        POSITION = 9, 83
        SIZE = 253, 16
        TOGGLE = TRUE
        SELECTED = TRUE
    }

    CHECKBUTTON = SDate_Cent
    {
        TITLE = 'Century (1990 vs 90)'
        POSITION = 9, 101
        SIZE = 187, 16
        TOGGLE = TRUE
    }
}
```

Note: The two static text objects both have an identifier of zero, and will therefore be assigned handles by UIMS. This should only be done for objects to which the application will never require access, since there is no way of discovering the values of these handles.

While your application is running:

```
SDATE = 320
SDATE.ORDER = 321
SDATE.MDY = 322
SDate.DMY = 323
SDate.YMD = 324
SDate.Sep = 325
SDate.Day = 326
SDate.Month = 327
SDate.Cent = 328

.
.
.

TITLE = "Short Date Format"
CALL CreateIncGroup(CONTEXT, ...
                    SDATE, ...
                    TITLE, ...
                    20, 14, ...
                    328, 125, ...
                    UIMS.BORDER, ...
                    "", ...
                    SDATE)

    CALL CreateText(CONTEXT, ...
                    0, ...
                    "Order:", ...
                    9, 18, ...
                    51, 12, ...
                    SDATE, ...
                    SDATE.ORDTXT)

    CALL CreateExGroup(CONTEXT, ...
                       SDATE.ORDER, ...
                       "", ...
                       93, 10, ...
                       220, 30, ...
                       UIMS.NONE, ...
                       SDATE, ...
                       SDATE.ORDER)
        CALL CreateOptionButton(CONTEXT, ...
                                SDATE.MDY, ...
                                "MDY", ...
                                5, 6, ...
                                0, 0, ...
                                SDATE.ORDER, ...
                                SDATE.MDY)
    CALL CreateOptionButton(CONTEXT, ...
                                SDATE.DMY, ...
                                "DMY", ...
                                78, 6, ...
                                0, 0, ...
                                SDATE.ORDER, ...
                                SDATE.DMY)
    CALL CreateOptionButton(CONTEXT, ...
                                SDATE.DMY, ...
                                "YMD", ...
                                151, 6, ...
                                0, 0, ...
                                SDATE.ORDER, ...
                                SDATE.DMY)
    CALL OptionButtonSelect(CONTEXT, SDATE.DMY, ERR)

CALL CreateText(CONTEXT, ...
                0, ...
                "Separator:", ...
                9, 42, ...
                84, 12, ...
                SDATE, ...
                SDATE.SEPTXT)
STYLE = UIMS.EBOX.RECT
JUST = UIMS.EBOX.LJUST
CALL CreateEditBox(CONTEXT, ...
                   SDATE.SEP, ...
                   98, 39, ...
                   27, 18, ...
                   STYLE, ...
                   "", ...
                   SDATE, ...
                   SDATE.SEP)

TITLE = "Day Leading Zero (07 vs 7)"
CALL CreateCheckButton(CONTEXT, ...
                       SDATE.DAY, ...
                       TITLE, ...
                       9, 65, ...
```

```
                        0, 0, ...
                        SDATE, ...
                        SDATE.DAY)
CALL CheckButtonSetToggle(CONTEXT, SDATE.DAY, TRUE, ERR)
CALL CheckButtonSelect(CONTEXT, SDATE.DAY, ERR)

TITLE = "Month Leading Zero (02 vs 2)"
CALL CreateCheckButton(CONTEXT, ...
                        SDATE.MONTH, ...
                        TITLE, ...
                        9, 83, ...
                        0, 0, ...
                        SDATE, ...
                        SDATE.MONTH)
CALL CheckButtonSetToggle(CONTEXT, SDATE.MONTH, TRUE, ERR)
CALL CheckButtonSelect(CONTEXT, SDATE.MONTH, ERR)

TITLE = "Century (1990 vs 90)"
CALL CreateCheckButton(CONTEXT, ...
                        SDATE.CENT, ...
                        TITLE, ...
                        9, 101, ...
                        0, 0, ...
                        SDATE, ...
                        SDATE.CENT)
CALL CheckButtonSetToggle(CONTEXT, SDATE.MCENT, TRUE, ERR)
```

Note: If you specify zero for both the width and height of an option or check button its size will be calculated automatically.

An Example Application: EditCtrl

This example application illustrates some ways of using controls in an application's main window. It you shows how to:

- Create a button bar to make commonly used commands more accessible.
- Use a TextEditor to provide multiple-line text entry and editing.
- Use a Text control to display status messages.

The EditCtrl application places a button bar at the top of the window and a status line at the bottom, and fills the rest of the client area with a TextEditor. It then monitors the size of the client area to ensure that the TextEditor always just fits and the status line is always at the bottom.

EditCtrl is an extension of the EditMenu application described in Chapter 8. To create the EditCtrl application, copy and rename the source files of the EditMenu application and then make the following changes.

1. Add new constant definitions to the header file.
2. Define TitledButton, TextEditor and Text resources.
3. Enable update and size messages.
4. Change the colour of the App window's client area.
5. Add the status-line Text contact as a Child of the main application window.
6. Add new variables.
7. Determine the sizes of the button bar and the status line, for use when positioning and resizing the TextEditor.
8. Add a UIMS.MSG.ENTER case to the message loop.
9. Add a UIMS.MSG.SIZE case to the message loop.
10. Add a UIMS.MSG.BUTTONPRESS case to the message loop.
11. Compile the resource file and the DATA/BASIC program.

Add New Constant Definitions

You will need identifiers for the additional resources defined in the resource script. These must be available to both the resource script and the DATA/BASIC source, so add the following to your header file:

```
EQUATE TxtEd1      TO 20
EQUATE Text1       TO 30

* Buttons

EQUATE SaveButton  TO 202
EQUATE CutButton   TO 204
EQUATE CopyButton  TO 205
EQUATE PasteButton TO 206
EQUATE UndoButton  TO 208
```

Ensure that the new header file is available on both the host and the PC.

Define New Resources

The application window will contain five TitledButton contacts for the button bar, a Text contact for the status line and a TextEditor contact. The TextEditor must allow you to scroll to any part of the text and will therefore need horizontal and vertical scroll bars. Note that the Text And TextEditor contacts are given arbitrary the sizes and positions; the correct values will be set within the application before these controls are mapped.

Add the following lines to the definition for Win1 in the file EDITCTRL.UCL:

```
  TITLEDBUTTON = SaveButton
  {
    POSITION = 0, 0
    SIZE = 96, 0
    TITLE = 'Save File'
  }

  TITLEDBUTTON = CutButton
  {
    POSITION = 106, 0
    SIZE = 96, 0
    TITLE = 'Cut'
  }

  TITLEDBUTTON = CopyButton
  {
    POSITION = 202, 0
    SIZE = 96, 0
    TITLE = 'Copy'
  }

  TITLEDBUTTON = PasteButton
  {
    POSITION = 298, 0
    SIZE = 96, 0
    TITLE = 'Paste'
    ENABLED = FALSE
  }

  TITLEDBUTTON = UndoButton
  {
    POSITION = 404, 0
    SIZE = 96, 0
    TITLE = 'Undo'
  }

  TEXTEDITOR = TxtEd1
  {
    POSITION = 0, 40
    SIZE = 700, 500
    STYLE = AUTOSCROLL, BORDER, HSCROLLBAR, VSCROLLBAR
    MAPPED = FALSE
  }

  TEXT = Text1
  {
    POSITION = 0, 480
    SIZE = 1000, 0
    CONTENT = '  Current status:'
    MAPPED = FALSE
  }
```

You will also need to increase the initial size of the main App window. Find the line which reads

```
SIZE = 500, 417
```

and change it to

```
SIZE = 700, 500
```

Enable Enter and Size Messages

When you start your application, size and enter messages will be disabled. To enable them, add the following code to the DATA/BASIC source after signing on to UIMS, but before loading the resources.

```
* Enable size and enter messages if not already enabled
CALL GetEventMask(CONTEXT, CONTEXT, EVENTMASK)
CALL BitTest(EVENTMASK, UIMS.EM.SIZE, ENABLED)
IF NOT(ENABLED) THEN EVENTMASK = EVENTMASK + UIMS.EM.SIZE
CALL BitTest(EVENTMASK, UIMS.EM.ENTER, ENABLED)
IF NOT(ENABLED) THEN EVENTMASK = EVENTMASK + UIMS.EM.ENTER
CALL SetEventMask(CONTEXT, CONTEXT, EVENTMASK, ERR)
```

There is no need to enable button-press messages, as the default event mask has these enabled.

Change the Window's Colour

To help distinguish the different parts of the application's window, the background of the main application window will be given a different colour. Add the following lines after to the DATA/BASIC source after the resources have been loaded, but before adding Win1 as a child of the application context.

```
* Change the colour of Win1's client area
CALL GetDrawrule(CONTEXT, CONTEXT, DRAWRULE1)
CALL DrawruleSetColour(CONTEXT, DRAWRULE1, UIMS.BLACK, UIMS.GREY,
ERR)
```

Add New Variables

You will need five new variables in the EditCtrl application. Add the following to the DATA/BASIC source code after the line which adds Win1 as a child of the application context:

```
* Mapped flag - resources are not yet mapped
MAPPED = 0

* Get heights of button bar and status line
CALL GetSize(CONTEXT, SaveButton, BUTTON.WIDTH, BUTTON.HEIGHT, ERR)
CALL GetSize(CONTEXT, Text1, STAT.WIDTH, STAT.HEIGHT, ERR)
```

The heights of the button bar and status line will be used to set the position of the status line, and the size and position of the TextEditor. BUTTON.WIDTH and STAT.WIDTH are in fact unused, but must be present in the calls to **GetSize**.

Add the Enter Case

Whenever the focus is moved from one contact to another, a **UIMS.MSG.ENTER** message is generated. In the EditCtrl application, the TextEditor must always have the focus, so that the user can enter and edit text. Enter messages will therefore be used to pass the focus to the TextEditor.

Note, however, that in the case of the TitledButton contacts, changing the focus too quickly can prevent the button-press message being generated. The focus is therefore only returned to the TextEditor once the button-press message has been received (see below).

Add the following to the CASE statement in the HANDLE.WIN1.MESSAGES subroutine.

```
CASE MSG.TYPE = UIMS.MSG.ENTER
  * If the contact is not a button, set the focus back to
  * the text editor.
  IF MSG.CONTACT < SaveButton THEN
    CALL SetContactFocus(CONTEXT, TxtEd1, ERR)
  END
```

Add the Size Case

A **UIMS.MSG.SIZE** message is generated whenever the size of a window is changed, and also when the window is displayed by giving it a parent. In EditCtrl, size messages will be used to set the position of the status line, and the size and position of the TextEditor. Add the following to the CASE statement in the HANDLE.WIN1.MESSAGES subroutine.

```
CASE MSG.TYPE = UIMS.MSG.SIZE
  * Win1's size has changed - so must TxtEd1's
  CALL Resize(CONTEXT, TxtEd1, INT(DATA1 / 65536), ...
              DATA2 - BUTTON.HEIGHT - STAT.HEIGHT, ERR)

  * Position the status line at the foot of the window
  CALL Move(CONTEXT, Text1, 0, DATA2 - STAT.HEIGHT, ERR)

  * If TxtEd1 and Text1 are not yet mapped -
  IF NOT(MAPPED) THEN
    * Move TxtEd1 to below the button bar
    CALL Move(CONTEXT, TxtEd1, 0, BUTTON.HEIGHT, ERR)
    * Map TxtEd1 and Text1
    CALL Map(CONTEXT, TxtEd1, ERR)
    CALL Map(CONTEXT, Text1, ERR)
    MAPPED = 1
  END
```

This sets the width of the TextEditor to be the same as that of the client area of Win1, as returned in the DATA1 parameter (note that this must be divided by 65536 to obtain the correct value). The required height for the TextEditor is calculated by subtracting the heights of the button bar and the status line from the height of the Win1 client area. Similarly, the status line is moved to the foot of the window, by subtracting its height from that of the Win1 client area and using this as the new horizontal position.

TxtEd1 and Text1 are created are created unmapped so that their positions and sizes can be set before they are displayed. When the first size message is received (generated when Win1 is made a child of the application context), the MAPPED variable will be zero; the TextEditor is therefore moved to its correct position, just below the button bar, and TxtEd1 and Text1 are then mapped (thus displaying them within Win1). The MAPPED variable is set to prevent these operations taking place on subsequent size messages.

Add the ButtonPress Case

The EditCtrl application is intended to demonstrate how to use controls in a window. Therefore, instead of performing tasks, the buttons at the top of the window display the message "Command not implemented!".

Note that once the message has been processed, the focus is passed to the TextEditor so that the user can continue editing.

In your DATA/BASIC source, add the following to the CASE statement in the HANDLE.WIN1.MESSAGES subroutine.

```
CASE MSG.TYPE = UIMS.MSG.BUTTONPRESS ;* user has clicked a button
      GOSUB HANDLE.WIN1.BUTTONS
```

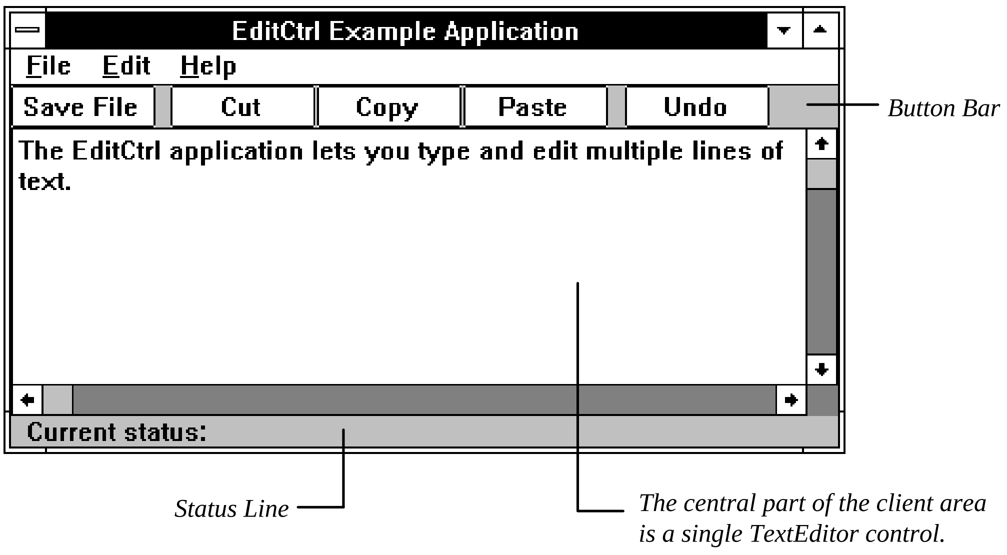
Then add the following subroutine at any convenient point:

```
***************************************************************
*
* SUBROUTINE: HANDLE.WIN1.BUTTONS
*
* PURPOSE: Process Win1 button press messages
*
* COMMENTS:
*   This routine takes action according to which button was
*   pressed by the user.
*
***************************************************************

HANDLE.WIN1.BUTTONS:
  BEGIN CASE ;* Switch on the contact in which the event occurred
*
```

```
  * File actions
  CASE MSG.CONTACT = SaveButton

    CALL CreateMessageBox(CONTEXT, ...
                          UIMS.INFO, ...
                          "EditCtrl Example Application", ...
                          "Command not implemented!", ...
                          "", ...
                          OK, ...
                          ERR)

  * Edit actions
  CASE MSG.CONTACT = CutButton OR ...
       MSG.CONTACT = CopyButton OR ...
       MSG.CONTACT = PasteButton OR ...
       MSG.CONTACT = UndoButton

    CALL CreateMessageBox(CONTEXT, ...
                          UIMS.INFO, ...
                          "EditCtrl Example Application", ...
                          "Command not implemented!", ...
                          "", ...
                          OK, ...
                          ERR)

  END CASE

  * Return the focus to TxtEd1
  CALL SetContactFocus(CONTEXT, TxtEd1, ERR)

RETURN
```

Compile

When you have made these changes, you can compile the resource script and DATA/BASIC program as described in Chapter 4 for the Generic application. When run, the application should look like this:


EditCtrl Example Application
File
Edit
Help
Save File
Cut
Copy
Paste
Undo
Button Bar
Current status:
Status Line
The central part of the client area is a single TextEditor control.

The EditCtrl application lets you type and edit multiple lines of text.

Figure 9-1. The EditCtrl Application

Whenever the application is active, the TextEditor will be given the focus, allowing you to enter and edit text. You can also select text with the keyboard and the mouse (see the *UIMS DATA/BASIC API, Reference Manual* for a full description of the TextEditor). If you change the size of the application's main window, the TextEditor will be resized to suit, and the status line will be repositioned at the foot of the window.

Chapter 10
Dialog Boxes

Dialog Boxes are windows that applications use to interact with the user. Typically a dialog box will contain one or more of the controls described in the previous chapter.

This chapter covers the following topics:

- What is a dialog box.
- Creating and using the different types of dialog box.
- Using controls in dialog boxes.

This chapter also explains how to create an example application which shows how to build and use a dialog box which contains controls.

What is a Dialog Box

A dialog box is a window that an application uses to display or prompt for information. Dialog boxes are typically used to prompt the user for the information needed to complete a command. A dialog box will contain one or more controls with which the user can type text, choose options, and direct the action of a particular command.

The About box in the Generic Application described in Chapter 3 is an example of a dialog box. It contains static text controls that provide information about the application, and a push button control that the user must use to close the dialog box and return to the main window.

A dialog box is normally displayed in response to a menu selection. For example, the Open and Save As commands on a File menu both require additional information to complete their tasks. Both display dialog boxes to prompt for a file name and a directory.

There are two types of dialog box: modal and modeless. These are described below.

Modal Dialog Boxes

A modal dialog box temporarily disables the parent window and forces the user to complete the requested action before continuing. Modal dialog boxes are particularly useful for gathering information your application requires in order to proceed. For example, most applications display a modal dialog box when the user chooses the Open command from the File menu. The application cannot proceed until the user has entered the name of the file to open.

There are two types of modal dialog box: application-modal and system-modal. An application-modal dialog box disables only the current application; the user can use other applications while the dialog box is displayed. A system-modal dialog box disables the complete user interface; the user can do nothing until he has responded to the dialog.

Modeless Dialog Boxes

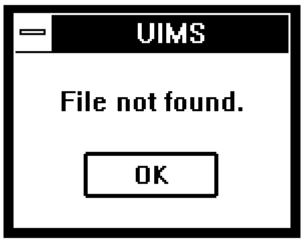
A modeless dialog box does not disable the parent window. This means that the user can continue to work with the application while the dialog box is displayed. For example, a drawing application might include a command that allows the user to choose a colour with which to fill graphic elements; the dialog box which appears when this command is selected might be modeless so that the colours of several elements could be changed without having to choose the Colour command each time.

Message Boxes

A message box is a special type of application-modal dialog box which is predefined as containing only a graphic icon, static text and command buttons. There are four pre-defined types of message box: Information, which has a single OK button; Warning, which has OK and Cancel, or Yes, No and Cancel buttons; Alert, which has either Retry and Cancel, or Abort, Retry and Ignore buttons; and Query, which displays a question mark and has 'OK' and 'Cancel', or 'Yes', 'No' and 'Cancel' buttons. The titles of the buttons can be changed if required. The graphic icon displayed depends on the type of message box selected. You can also define your own combination of icon and buttons.

Using a Dialog Box

You create a dialog box in much the same way as any other contact: either while your application is running by calling the **CreateDlgBox** subroutine, or in a separate resource script. Both the following examples create a dialog box containing some static text and an OK button.



In the resource script:

```
EQUATE Fnf_Dlg TO 830
EQUATE Fnf_OK TO 831
DIALOGBOX = Fnf_Dlg
{
    TITLE = 'UIMS'
    POSITION = 63, 42
    SIZE = 148, 92
    STYLE = SYSMENU ❶

    TEXT = 0 ❷
    {
        CONTENT = 'File not found.'
        POSITION = 28, 31
        SIZE = 116, 12
    }

    TITLEDBUTTON = Fnf_OK
    {
        TITLE = 'OK'
        POSITION 46, 55
        SIZE = 60, 18
    }
}
```

In this example:

❶ The **SYSMENU** style gives the dialog box a system menu.

❷ An identifier of zero asks UIMS to assign a handle to the text contact.

While your application is running:

```
FNF.DLG = 830
FNF.OK =  831

TITLE = "UIMS"
STYLE = UIMS.WIN.SYSMENU
CALL CreateDlgBox(CONTEXT, FNF.DLG, TITLE, 63, 42, 148, 92, ...
                    STYLE, ""❶, FNF.DLG)
CALL SetMapped(CONTEXT, FNF.DLG, FALSE, ERR) ❷

CONTENT = "File not found."
CALL CreateText(CONTEXT, 0❸, CONTENT, 18, 31, 116, 12, FNF.DLG, ...
                  FNF.TXT)

CALL CreateTitledButton(CONTEXT, FNF.OK, "OK", 46, 55, 60, 18, ...
                          FNF.DLG, FNF.OK)

CALL AddChild(CONTEXT, WIN1, -1, FNF.DLG, ERR)
```

In this example:

❶ The dialog box is initially created without a parent so that it will not be displayed.

❷ This line sets the dialog upmappable, so that it can be attached to its parent without being displayed. Note that its children do not need to be set unmappable.

❸ An identifier of zero asks UIMS to assign a handle to the text contact.

When you first create a dialog box it is always application-modal; if you want to change its mode you must call the **DlgBoxSetMode** subroutine.

When you need the dialog box, there are several ways in which you can display it:

- You could attach the dialog box to its parent, but set it unmappable (by calling the **SetMapped** subroutine) so that it will not normally be displayed. When you need the box you simply change it to mappable and change it back again when you have finished.

- You could create or load the dialog box as an orphan and attach it to its parent window when needed. When you have finished with the dialog you would remove it from its parent's list of children.

- You could create or load the dialog box resource only when it is needed and destroy it again when you have finished.

The first of these is the simplest and quickest method to use.

Using Controls in Dialog Boxes

You use controls in a dialog box in much the same way as in an **AppWindow** or a **ChildWindow**. You will need a routine to handle messages generated within the window and call this from the window case statement in your application's message loop.

There are three ways in which the user can leave a dialog box:

- By operating one of the titled buttons.

- By pressing return while in one of the other controls, and thus actioning the default titled button.

  Note: Within a dialog box, you do not need to specially create a default titled button. You can call the **DlgBoxSetDefButton** subroutine to do this for you and the dialog box routines will then change the button borders as necessary.

- If the dialog box has a system menu, the user can choose the Close command.

In the first two cases, a **UIMS.MSG.BUTTONPRESS** message is generated, and this specifies which button was operated. In the third case, a **UIMS.MSG.CLOSE** message is generated, specifying the dialog box as the window to be closed. Note that in either case, the event window is the dialog box itself; events which occur within a dialog box can therefore be handled separately from those which occur in other windows. The following example illustrates this:

```
LOOP UNTIL USER.WANTS.TO.EXIT DO

    CALL GetMsg(0, ...
                MSG.CONTEXT, ...
                MSG.WINDOW, ...
                MSG.CONTACT, ...
                MSG.TYPE, ...
                TIMESTAMP, ...
                DATA1, ...
                DATA2, ...
                DATA3, ...
                DATA4)

    BEGIN CASE
    CASE MSG.WINDOW=WIN1

        .
        . REM handle messages for WIN1
        .

    CASE MSG.WINDOW=DLG1

        *   handle messages for the dialog box
        *   we are only interested in button-press and close messages
        BEGIN CASE

        CASE MSG.TYPE=UIMS.MSG.BUTTONPRESS
            GOSUB HANDLE.DLG1.BUTTONS

        CASE MSG.TYPE=UIMS.MSG.CLOSE
            *   for a close message, simply unmap the dialog box
            CALL UnMap(CONTEXT, DLG1, ERR)

        END CASE

    END CASE

REPEAT

    .
    .
    .

HANDLE.DLG1.BUTTONS:
    BEGIN CASE

    CASE MSG.CONTACT = DLG1.CANCEL
        *   if the cancel button was operated, unmap the dialog box
        CALL UnMap(CONTEXT, DLG1, ERR)
    CASE MSG.CONTACT = DLG1.OK
        .
        . REM carry out requested action
        .

    .
    . REM handle any other titled buttons
    .

    END CASE
RETURN
```

Using a Message Box

A message box can only be created while your application is running, by calling the **CreateMessageBox** subroutine. The message box is always displayed immediately and destroyed when the user operates one of its buttons.

When creating a message box, you must provide the following:

- The handle of the application context.
- A value that specifies –
    1. The number of buttons.
    2. The type of icon (Information, Warning, Alert or Query) to be displayed in the message box.
    3. Which button will be the default.
- The title of the message box, if any.
- The text of the message.
- A list of titles for the buttons.
- Variables in which to return the response to the message (which button was operated) and a completion code.

Style Parameter

The style of the message box is determined by a value that is made up of three different elements:

1. A value from 1 to 3 that determines the number of buttons.
2. A value from 16 to 64 in steps of 16, that determines the type of icon to be displayed in the message box. You can use any one of four icons:

Value	Purpose	Icon
16	Information	ℹ
32	Warning	❗
48	Alert	🛑
64	Query	❓

3. A value from 0 to 512 in steps of 256, that determines which button is the default.

For example, the value 34 (2 + 32 + 0) specifies two buttons, a Warning icon and the first button as the default, while 323 (3 + 64 + 256) specifies three buttons, a Query icon and the second button as the default.

Button Titles

The button titles must be listed in a dynamic array, with one title in each attribute. Note that if you specify a null string for any of the titles, the appropriate button will be given a pre-determined default title. This is particularly useful when using the pre-defined message box styles (see below).

Pre-defined Styles

A number of pre-defined styles are available. These specify the number of buttons, the icon used and the titles of the buttons. You must specify the required pre-defined style instead of the number of buttons and the icon, and use a null string instead of the array of button titles. For example, the style **UIMS.ALERT2** specifies a message box with an Alert icon and two buttons labelled "Retry" and "Cancel":

If you wish, you can combine a pre-defined style with a default button specification. For example:

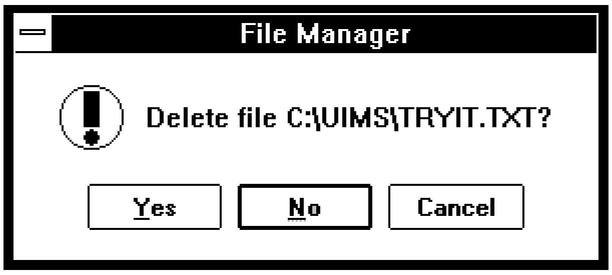
```
UIMS.ALERT2 + 256
```

specifies the same message box as above, but with the second, "Cancel" button as the default.

If necessary, you can use a button title array to replace one or more of the pre-defined titles.

Example

The following example creates a warning message which might be displayed when you attempt to delete a file:



Delete file C:\UIMS\TRYIT.TXT?

```
FILE = "C:\UIMS\TRYIT.TXT"
MESSAGE = "Delete file ":FILE:"?"
TYPE = 3 + 32 + 256 ❶
TITLE = "File Manager"
BUTTONS = "Yes":AM:"No":AM:"Cancel" ❷
CALL CreateMessageBox(CONTEXT, ...
                      TYPE, ...
                      TITLE, ...
                      MESSAGE, ...
                      BUTTONS, ...
                      RESPONSE, ...
                      ERR)
```

In this example:

❶ This parameter specifies a message box with three buttons (3) and a warning icon (32), in which the second button is the default.

❷ This parameter defines the titles of the buttons in the message box.

An Example Application: SendMsg

This example application shows you how to create and use an application-modal dialog box to provide access to one of the standard REALITY commands: MESSAGE. The application has a Utilities menu containing a Send Message command. This displays the dialog box shown in Figure 10-1.

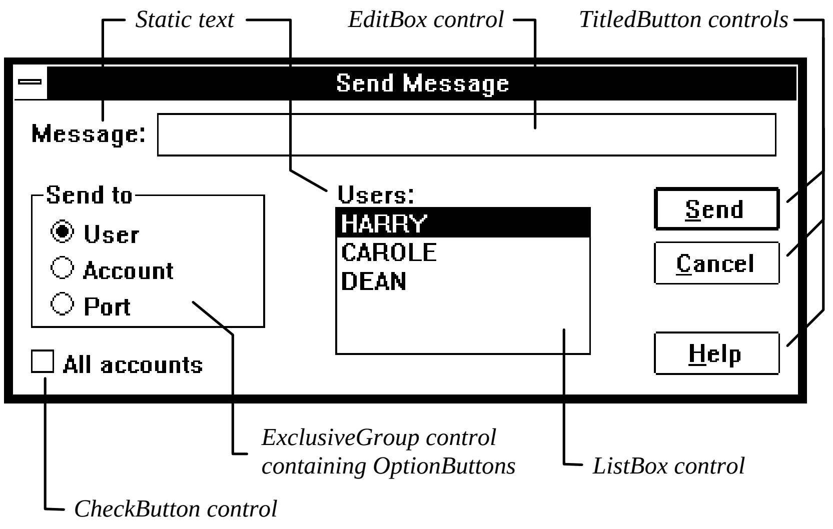


Figure 10-1. The Send Message Dialog Box

The dialog box contains the following controls:

- An EditBox control in which the user types the message to be sent.
- An ExclusiveGroup containing three OptionButton controls – these allow the user to choose whether to send the message to another user, to a specified account or to a specified port on the host system.
- A ListBox control which lists the users that are currently logged on, or the accounts or ports that are currently in use. The contents of this control change as the user chooses from the ExclusiveGroup.
- A CheckButton control which allows the user to send the message to every account on the host system.

- A default TitledButton control labelled "Send". This lets the user tell the application to send the message.
- A button labelled "Cancel" that lets the user cancel the Send Message command.
- A button labelled "Help" that in a real application would offer the user help on using the Send Message command. In the example application, this is not implemented and a message box is displayed instead.
- Two static Text controls that label the EditBox and the ListBox.

To create the SendMsg application, copy and rename the source files for the Generic application as described in Chapter 4. Then make the following changes:

1. Add new constant definitions to the header file.
2. Define a Utilities menu and a Send Message menu item.
3. Define the Send Message dialog box and its controls.
4. Modify the menu item case in the message loop.
5. Add the Send Message window case to the main message loop.
6. Add a HANDLE.DLG1.MESSAGES subroutine.
7. Add subroutines to support the message-handling routines.
8. Compile the resource file and the DATA/BASIC program.

Add New Constant Definitions

You will need identifiers for the additional resources defined in the resource script. These must be available to both the resource script and the DATA/BASIC source, so add the following to your header file:

```
* Utilities menu

EQUATE UtilSendMsg TO 101


* Dialog Box

EQUATE Dialog1      TO 200
EQUATE Text1        TO 210
EQUATE Edit1        TO 211
EQUATE Text2        TO 220
EQUATE List1        TO 222
EQUATE ExGroup      TO 230
EQUATE Option1      TO 231
EQUATE Option2      TO 232
EQUATE Option3      TO 233
EQUATE Check1       TO 240
EQUATE OKButton     TO 250
EQUATE CancelButton TO 251
EQUATE HelpButton   TO 252
```

Ensure that the new header file is available on both the host and the PC.

Define Resources

In order to use the Send Message command, the user must be able to select it from a menu. Find the definition of Win1's menu bar in the resource script and add the following after the opening brace:

```
    MENU = 0
    {
      TITLE = '&Utilities'
      CHILDREN = 'Send &Message...' = UtilSendMsg
    }
```

Although the dialog box will not be displayed until the user selects the Send Message command, it will be created in the resource script as a child of Win1. To prevent it being displayed all the time, its Mapped attribute will be set to FALSE. Then, when the dialog box is required all that is necessary is to Map it.

Add the following to the resource script, just before the final closing brace of the definition for Win1:

```
  DIALOGBOX = Dialog1
  {
    TITLE = 'Send Message'
    POSITION = 235, 300
    SIZE = 698, 549
    STYLE = CLOSABLE, MOVABLE ❶
    MAPPED = FALSE
```

```
TEXT = Text1
{
  CONTENT = 'Message:'
  POSITION = 16, 29
  SIZE = 0, 0
}

EDITBOX = Edit1
{
  POSITION = 125, 21
  SIZE = 538, 69
  STYLE = BORDER
  MASK = '' ❷
}

EXCLUSIVEGRP = ExGroup
{
  TITLE = 'Send to' ❸
  POSITION = 16, 126
  SIZE = 203, 235
  BORDER = BORDER

  OPTIONBUTTON = Option1
  {
    TITLE = 'User'
    POSITION = 16, 5 ❹
    SIZE = 0, 0
    SELECTED = TRUE
  }

  OPTIONBUTTON = Option2
  {
    TITLE = 'Account'
    POSITION = 16, 63 ❹
    SIZE = 0, 0
  }

  OPTIONBUTTON = Option3
  {
    TITLE = 'Port'
    POSITION = 16, 120 ❹
    SIZE = 0, 0
  }
}

CHECKBUTTON = Check1
{
  TITLE = 'All accounts'
  POSITION = 16, 383
  SIZE = 0, 0
}

TEXT = Text2
{
  CONTENT = 'Users:'
  POSITION = 281, 126
  SIZE =  216, 0
}

LISTBOX = List1
{
  POSITION = 281, 171
  SIZE = 219, 228
  CONTROLS = NONE ❺
}

TITLEDBUTTON = SendButton
{
  TITLE = '&Send'
  POSITION = 556, 137
  SIZE = 110, 69
}

TITLEDBUTTON = CancelButton
{
  TITLE = '&Cancel'
  POSITION = 556, 223
  SIZE = 110, 69
}

TITLEDBUTTON = HelpButton
{
  TITLE = '&Help'
  POSITION = 556, 366
```

```
      SIZE = 110, 69
    }

    DEFBUTTON = SendButton ❻
  }
```

❶ The user will be able to move the dialog box around the screen and to close it from the system menu, as well as with the Cancel button.

❷ In the definition of the EditBox control, the MASK attribute is intended for future use. It must be included, however, and should be set to a null string.

❸ The ExclusiveGroup control has its own title, and does not need a static text control.

❹ The position of each contact must be specified relative to its parent, so the three OptionButton controls must be positioned relative to the ExclusiveGroup. All other controls are positioned relative to the DialogBox window.

❺ The style of the list box is set to NONE. This will allow only one item to be selected at a time.

❻ The Send button is made the default.

Note that there is no need to set the mode of the DialogBox, since the default is application-modal.

Modify the Menu Item Case

The SendMsg application has a Utilities menu containing the Send Message command, so you will need add this to the message loop. Edit the DATA/BASIC source code and add the following to the CASE structure in the HANDLE.WIN1.MENU subroutine:

```
* Send Message command
CASE MSG.CONTACT = UtilSendMsg
  * Return the dialog to its default state
  GOSUB SETUP.DIALOG1

  * Now display the dialog box
  CALL Map(CONTEXT, Dialog1, ERR)

  * Give the message edit box the focus
  CALL SetContactFocus(CONTEXT, Edit1, ERR)
  * Select the whole of any previous message
  CALL EditBoxSetSelected(CONTEXT, Edit1, 0, 0, TRUE, ERR)
```

The SETUP.DIALOG1 subroutine sets the dialog to its default state. This is necessary because each time the dialog is used, there might be different users logged on.

The **Map** subroutine displays the dialog box. Once this has been done, the focus can be set to the Message edit box; if this is not done, the user will have to click in the edit box before starting to type (note that this cannot be done until the dialog box has been mapped, as it is not possible to give the focus to an unmapped contact). Finally, the whole of any text already in the edit box is selected; this will allow to user to delete it by simply starting to type a new message.

Add the Dialog1 Case

For messages generated while the Send Message dialog box is displayed, the event window parameter will be the handle of the dialog box. You will therefore need to add the following to the CASE structure in the main message loop:

```
  CASE MSG.WINDOW = Dialog1
    GOSUB HANDLE.DLG1.MESSAGES
```

Then add the subroutine HANDLE.DLG1.MESSAGES at any convenient point in your DATA/BASIC source code:

```
***************************************************************
*
* SUBROUTINE: HANDLE.DLG1.MESSAGES
*
* PURPOSE: Process messages for the Send Message dialog box
*
* COMMENTS:
*   The dialog box is closable, so Close messages must be processed.
*   Otherwise, only button-press messages are processed.
*
***************************************************************

HANDLE.DLG1.MESSAGES:
  BEGIN CASE ;*   switch on the type of message
*
  CASE MSG.TYPE = UIMS.MSG.BUTTONPRESS
*
    BEGIN CASE ;* Switch on the contact in which the event occurred
*
    * Send button
    CASE MSG.CONTACT = SendButton
  * Get the message to be sent
  CALL EditBoxGetContent(CONTEXT, Edit1, MESSAGE, VALID, ERR)
  IF MESSAGE = "" THEN

    * If no message, beep and set focus to the message edit box
    CALL SoundSpeaker(1024, 50, 1, 0, ERR)
    CALL SetContactFocus(CONTEXT, Edit1, ERR)

  END ELSE

    * Otherwise, send the message
    GOSUB SEND.MSG

    * Hide the dialog box
    CALL UnMap(CONTEXT, Dialog1, ERR)

  END

* Cancel button
CASE MSG.CONTACT = CancelButton
  * Hide the dialog box
  CALL UnMap(CONTEXT, Dialog1, ERR)

* Help button
CASE MSG.CONTACT = HelpButton
  * Help is not implemented, so display a message
  CALL CreateMessageBox(CONTEXT, ...
                        UIMS.INFO, ...
                        "Dialog Box Example Application", ...
                        "Command not implemented!", ...
                        "", ...
                        OK, ...
                        ERR)

CASE MSG.CONTACT = Check1
  * Find out the new state of the All Accounts button
  CALL CheckButtonGetSelected(CONTEXT, Check1, CHECK1.SEL)
  ENABLED = NOT(CHECK1.SEL)

  * If selected, the Ex-group and the list box must be
  * disabled; otherwise they are enabled.
  CALL SetEnabled(CONTEXT, ExGroup, ENABLED, ERR)
  CALL SetEnabled(CONTEXT, Option1, ENABLED, ERR)
```

```
      CALL SetEnabled(CONTEXT, Option2, ENABLED, ERR)
      CALL SetEnabled(CONTEXT, Option3, ENABLED, ERR)
      CALL SetEnabled(CONTEXT, List1, ENABLED, ERR)

    CASE MSG.CONTACT = Option1 OR ...
         MSG.CONTACT = Option2 OR ...
         MSG.CONTACT = Option3
      * New destination-type selection - fetch the appropriate list
      DEST.TYPE = MSG.CONTACT
      GOSUB GET.DEST

    END CASE

  * Close item on the system menu
  CASE MSG.TYPE = UIMS.MSG.CLOSE
    * Hide the dialog box
    CALL UnMap(CONTEXT, Dialog1, ERR)

  END CASE

RETURN
```

This routine handles two types of message: **UIMS.MSG.BUTTONPRESS** messages from the three titled buttons, the three option buttons and the check button; and **UIMS.MSG.CLOSE** messages generated by selecting Close from the system menu, or by double clicking the system menu box.

Send Button

When the Send button is operated, the application uses **EditBoxGetContent** to fetch the message that the user has typed. If there is no message, an alarm is sounded by calling the **SoundSpeaker** subroutine and the focus is then moved to the Message edit box.

If there is a message to send, the SEND.MSG subroutine is called to send the message and, when it has been sent, the dialog box is hidden.

Cancel Button

If the Cancel button is selected, the dialog box is simply hidden.

Help Button

There is no on-line help available for the SendMsg application, so a message is displayed, saying that the command is not implemented.

"All Accounts" Check Button

If the user operates the All Accounts check button, the application calls the **CheckButtonGetSelected** subroutine to find out the new state of the button, and then uses **SetEnabled** to enable or disable, as appropriate, the Send To group and the Destination list box. Note that in the case of the exclusive group, each option button must be enabled or disabled separately; if this is not done, when the group is disabled, there will be no indication to the user that these buttons cannot be used.

"Send To" Exclusive Group

When the user selects one of the option buttons in this group, the application changes contents of the Destination list box. First it sets the variable DEST.TYPE to the handle of the selected button, and then it calls the GET.DEST subroutine to get an up-to-date list of users, accounts or ports, as appropriate.

Close Messages

Selecting Close from the system menu, or double-clicking the system menu box, is the same as operating the Cancel button – the dialog box is simply hidden.

Add Support Subroutines

You need to add several subroutines to your DATA/BASIC source file to support the dialog box operations. These are:

SETUP.DIALOG1	This sets the Send Message dialog box to its default state.
GET.DEST	This fetches the current list of users, accounts or ports and displays it in the list box.
SEND.MSG	This sends the message entered by the user to selected user, account or port, or to all accounts.

SETUP.DIALOG1 Subroutine

The subroutine calls **CheckButtonDeselect** to deselect the All Accounts check button, and then uses the **Enable** subroutine to enable the Send To exclusive group and its option buttons, and the list box. The Users option button is then selected.

Next the DEST.TYPE variable is set to the handle of the Users option button and the GET.DEST subroutine is called; this fills the List1 list box with the list of current users.

Finally, the Send titled-button is made the default.

```
***************************************************************
*
* ROUTINE: SETUP.DIALOG1
*
* PURPOSE:
*   Sets the Send Message dialog box to its default state.
*
***************************************************************

SETUP.DIALOG1:
  * Deselect the All Accounts option and enable the Ex-group and
  * list box
  CALL CheckButtonDeselect(CONTEXT, Check1, ERR)
  CALL Enable(CONTEXT, ExGroup, ERR)
  CALL Enable(CONTEXT, Option1, ERR)
  CALL Enable(CONTEXT, Option2, ERR)
  CALL Enable(CONTEXT, Option3, ERR)
  CALL Enable(CONTEXT, List1, ERR)

  * Select the Users option
  CALL OptionButtonSelect(CONTEXT, Option1, ERR)

  * Get the list of current users and put it in the list box
  DEST.TYPE = Option1
  GOSUB GET.DEST

  * Make the Send button the default
  CALL DlgBoxSetDefButton(CONTEXT, Dialog1, SendButton, ERR)

RETURN
```

GET.DEST Subroutine

This subroutine fetches the list of current users, or of the accounts or ports that are in use, and displays the result in the Destination list box.

A list of users, accounts and ports is obtained by using the PERFORM statement to execute the TCL LISTU command. The data is returned in the form of a table, consisting of a header and a line of information for each port that is in use. The columns of this table contain:

1. The port number (characters 1 to 4).

2. The user-id of the user logged on via this port (characters 6 to 16).

3. The account being used by that user (characters 17 to 37).

The remaining columns contain additional information which is not needed by the SendMsg application.

The required list is built-up by extracting the appropriate columns from each line of the table (ignoring the header and the summary information at the end), and appending these to a dynamic array, DEST.LIST. When the list is complete, all spaces and the trailing attribute mark are removed, (if the trailing attribute mark is not removed, the list will have an extra, null item when displayed in the list box). The previous contents of the list box are then removed by calling **ListBoxRemoveContents** and the new list is inserted with **ListBoxAddContents**.

Next, the title of the list box (that is, the contents of the Text2 static text control) is changed to correspond to the contents of new list, and finally, the **ListBoxAddSelection** subroutine is called to select the first item in the list.

```
****************************************************************
*
* ROUTINE: GET.DEST
*
* PURPOSE:
*   Fetches the Ports that are currently in use, or the Users or
*   Accounts that are currently using them. The result is displayed
*   in the List1 list box.
*
* COMMENTS:
*   The type of destination is specified by the DEST.TYPE variable.
*   The current list is available in the DEST.LIST dynamic array.
*
****************************************************************

GET.DEST:
  * get the list of users
  PERFORM "LISTU" CAPTURING SYSDATA

  * initialise dynamic array
  DEST.LIST = ""

  * separate out the required info: ports, users or accounts
  FOR N = 6 TO DCOUNT(SYSDATA, CHAR(254))-2
    * ignore this port and entries with no port number
    IF SYSDATA<N>[1,1] <> "*" AND SYSDATA<N>[1,4] <> SPACE(4) THEN
      BEGIN CASE
      CASE DEST.TYPE = Option1
```

```
            DEST.LIST<-1> = SYSDATA<N>[6,10]
          CASE DEST.TYPE = Option2
            DEST.LIST<-1> = SYSDATA<N>[17,20]
          CASE DEST.TYPE = Option3
            DEST.LIST<-1> = SYSDATA<N>[1,4]
          END CASE
        END
      NEXT N

      * remove spaces from the list
      DEST.LIST = TRIM(DEST.LIST, " ", "A")
      * remove the trailing attribute mark
      DEST.LIST = TRIM(DEST.LIST, CHAR(254), "T")

      * Put the list into the list box
      CALL ListBoxRemoveContents(CONTEXT, List1, 0, -1, ERR)
      CALL ListBoxAddContents(CONTEXT, List1, -1, DEST.LIST, ERR)

      * Set the title of the list box
      BEGIN CASE
      CASE DEST.TYPE = Option1
        CALL TextSetContent(CONTEXT, Text2, "Users:", ERR)
      CASE DEST.TYPE = Option2
        CALL TextSetContent(CONTEXT, Text2, "Accounts:", ERR)
      CASE DEST.TYPE = Option3
        CALL TextSetContent(CONTEXT, Text2, "Ports:", ERR)
      END CASE

      * Select the first item
      CALL ListBoxAddSelection(CONTEXT, List1, 0, ERR)

    RETURN
```

SEND.MSG Subroutine

This subroutine sends the message entered by the user to the selected destination or destinations.

First, the application finds out whether the message is to be sent to all accounts by calling **CheckButtonGetSelected** to interrogate the All Accounts check button. If this option is selected, the CMND variable is set to an asterisk ("*").

If All Accounts is not selected, the type of destination is obtained by calling **ExGroupGetSel** to find out which of the Send To option buttons is selected. Next, the user-id, account name or port number is obtained by calling **ListBoxGetSelections** to find out which item in the list box is selected, and then setting the CMND variable to the corresponding attribute from the DEST.LIST array. The appropriate destination-type character is then added to the beginning of the CMND string: an exclamation mark ("!") for a port number; or an "@" character for a user-id. No prefix is required for an account name.

The message is then sent by using the PERFORM statement to execute the TCL MESSAGE command, using the CMND variable as the first parameter. The MESSAGE variable contains the message to be sent.

The screen output that results from sending the message is captured in the RESULT variable. After trimming off the leading and trailing attribute marks, the section following the first attribute mark is extracted. If the MESSAGE command was successful, the resulting string begins with the number of messages sent. This is tested to see whether one or more messages were sent, and the string is changed to read "MESSAGE" or "MESSAGES" as appropriate (if the MESSAGE command was unsuccessful, the error string which resulted remains unchanged). The resulting status message is then displayed in a message box.

```
***************************************************************
*
* ROUTINE: SEND.MSG
*
* PURPOSE: Send a message to another user, account or port
*
* COMMENTS:
*   SEND.TO - type of destination (user, account or port).
*   DEST.LIST - list of destinations.
*   MESSAGE - message to send.
*
***************************************************************

SEND.MSG:
  * Is the message to go to all accounts?
  CALL CheckButtonGetSelected(CONTEXT, Check1, SEND.TO)
  IF SEND.TO = TRUE THEN
    CMND = "*"
  END ELSE
    * If not, get the type of destination
    CALL ExGroupGetSel(CONTEXT, ExGroup, SEND.TO)

    * Get the destination
    CALL ListBoxGetSelections(CONTEXT, List1, LIST1.SEL, ERR)
    * set up destination parameter
    CMND = DEST.LIST<LIST1.SEL>
    BEGIN CASE
    CASE SEND.TO = Option1
      CMND = "@":CMND
    CASE SEND.TO = Option3
      CMND = "!":CMND
    END CASE
  END

  * send the message
  PERFORM "MESSAGE ":CMND:" ":MESSAGE CAPTURING RESULT

  * process the resulting completion message
  RESULT = TRIM(RESULT, CHAR(254), "B")
  RESULT = RESULT[INDEX(RESULT, CHAR(254), 1)+1, LEN(RESULT)]
  IF RESULT[1,2] = "1 " THEN
    RESULT = RESULT[1, INDEX(RESULT, "(S)", 1)-1]:...
             RESULT[INDEX(RESULT, ")", 1)+1,LEN(RESULT)]
  END ELSE
    RESULT = CONVERT(RESULT, "()", "")
  END

  * display the result in a message box
  CALL CreateMessageBox(CONTEXT, ...
                        UIMS.INFO, ...
                        "Send Message", ...
                        RESULT, ...
                        "", ...
                        OK, ...
                        ERR)

RETURN
```

Compile

When you have made these changes, you can compile the resource script and DATA/BASIC program as described in Chapter 4 for the Generic application. When you run SendMsg and select the Send Message command from the Utilities menu, you will see a dialog box similar to that shown in Figure 10-1. Type a message in the edit box, select a destination and then click Send to send the message:

Chapter 11
The Clipboard

The UIMS **Clipboard** object provides access to the Windows clipboard, thus allowing data to be copied and moved within a UIMS application, and exchanged between UIMS applications and other Windows applications.

This chapter covers the following topics:

- Copying text to the clipboard.
- Pasting text from the clipboard.
- Controlling the Cut, Copy and Paste items on the Edit menu.

It also describes an example application that illustrates many of the concepts explained in this chapter.

Using the Clipboard

The clipboard provides a temporary storage location for data which is being transferred from one place to another. Typically, the user will select the data to be transferred and then either cut or copy it to the clipboard. The destination for the data, in the same or a different application, is then selected and the data is pasted in.

The user normally carries out these operations by means of either the Edit menu commands, or by standard key-combinations. To add an Edit menu to your application, follow the steps described in Chapter 8.

At present, the UIMS **Clipboard** object only supports data in ASCII text format. However, other formats will be supported in the future, and you must therefore specify the format required when using the clipboard.

Copying Text to the Clipboard

There are several ways of copying text to the clipboard:

1. You can use the **ClipboardSetContent** subroutine to copy a specified text string to the clipboard.

2. You can use the **Copy** subroutine to copy text to the clipboard from an **EditBox** or **TextEditor** contact.

3. You can use the **Cut** subroutine to delete text from an **EditBox** or **TextEditor** contact, and transfer it to the clipboard.

The first of these is the simplest to use. For example:

```
CLIP = "Text for transfer" ❷
CALL ClipboardSetContent("TEXT"❶, CLIP❷, LEN(CLIP)❸, ERR❹)
```

❶ This parameter specifies the format of the data which is being placed on the clipboard. At present only "TEXT" format is supported.

❷ This parameter is the text to be placed on the clipboard.

❸ This parameter must specify the length of the data.

❹ This parameter specifies the name of a variable in which a completion code can be returned.

If you need to transfer data to the clipboard from an edit control, you must specify the start and end positions of the data to be transferred. For example:

```
CALL Copy(CONTEXT, TextEd1❶, 5❷, 1❷, 15❸, 1❸, ERR)
```

❶ The second parameter is the handle of the edit control; in this case a text editor.

❷ These two parameters specify the position of the first character to be copied. The third parameter is the character number, counting from the start of the line specified in the fourth parameter.

Note that lines number from zero, so setting the fourth parameter to 1 specifies the second line. If the contact were an **EditBox**, this would be ignored, since edit boxes can have only one line.

❸ These two parameters specify the position of the last character to be copied.

In order to use the **Cut** and **Copy** subroutines, you will need to keep track of any text selections made by the user within your edit controls. This can be done by monitoring **UIMS.MSG.SELECT** messages. For example:

```
CASE MSG.TYPE = UIMS.MSG.SELECT
   BEGIN CASE
   CASE MSG.CONTACT = TextEd1
      TEXTED1.START.CHAR = DATA2
      TEXTED1.START.LINE = INT(DATA1 / 65536) ❶
      TEXTED1.END.CHAR = DATA4
      TEXTED1.END.LINE = INT(DATA3 / 65536) ❶

   END CASE
```

❶ The values returned in the DATA1 and DATA3 parameters are offset by 65536, and must be adjusted to obtain the true value.

When the user selects the Cut or Copy menu item, these values can be used in the appropriate subroutine call. If all four values are zero, no text is selected.

Notes:

1. You will only need to use the **Cut** and **Copy** subroutines when processing the Cut and Copy menu items. When the cut (SHIFT+DELETE) and copy (CTRL+INSERT) key

combinations are used in an edit box or a text editor, the data is automatically transferred to the clipboard by UIMS.

2. If you prefer, you can cut or copy the text that is currently selected in the edit control. To do this, set all four start- and finish-position parameters to -1. Alternatively you can cut or copy all the text by setting these parameters to 0.

Pasting Text from the Clipboard

There are two ways of fetching data from the clipboard:

1. The **ClipboardGetContent** subroutine can be used to place the data in a variable. The contents of the variable can then be displayed in the client area.

2. The **Paste** subroutine can be used to paste text into an edit box or a text editor.

You should let the user paste only if there is text available on the clipboard. You can find out if there is text on the clipboard by calling the **ClipboardGetSize** subroutine.

The following example shows how to fetch text from the clipboard and display it in the client area:

```
* Find out if there is text available
CALL ClipboardGetSize("TEXT", CLIP.SIZE) ❶

* If there is, fetch the text
IF CLIP.SIZE THEN
  CALL ClipboardGetContent("TEXT", CLIP.DATA, CLIP.LENGTH)❷
END

* Display it in the client area
CALL DrawTextString(CONTEXT, Win1, CLIP.DATA, 10, 10, ERR)
```

In this example:

❶ The **ClipboardGetSize** subroutine returns the amount of data on the clipboard. The first parameter is the required data format, and the second a variable in which to return the size.

❷ When calling the **ClipboardGetContent** subroutine, you must specify the data format (first parameter) and provide variables in which the data and the data length can be returned.

When you use the **Paste** subroutine to transfer text from the clipboard to an edit control, you must specify the position in the control at which to insert the text. For example:

```
CALL Paste(CONTEXT, TextEd1❶, 5❷, 1❷, ERR)
```

❶ This parameter is the handle of the edit control.

❷ These two parameters specify the position at which the text will be inserted. The third parameter is the character number, counting from the start of the line specified in the fourth parameter. Note that lines number from zero, so setting the fourth parameter to 1 specifies the second line. If the contact were an **EditBox**, this would be ignored, since edit boxes can have only one line.

If you set both these parameters to -1, the text is inserted at the current cursor position, replacing any selected text.

Note: You will only need to use the **Paste** subroutine when processing the Paste menu item. When the paste (SHIFT+INSERT) key combination is used in an edit box or a text editor, the data is automatically transferred from the clipboard by UIMS.

Controlling the Edit Menu

You should only allow the user to transfer data to and from the clipboard if there is data available for transfer. This means that you should only enable the Cut and Copy commands on the Edit menu when the user has made a selection. Similarly, the Paste command should only be enabled if there is data available on the clipboard.

The following sections describe how to control the items on the Edit menu when you are using edit controls.

The Cut and Copy Items

Since a Select message is generated whenever the user makes a text selection in an edit control, these messages can be used to enable and disable the Cut and Copy menu items. For example:

```
CASE MSG.TYPE = UIMS.MSG.SELECT
   BEGIN CASE
   CASE MSG.CONTACT = TextEd1
      TEXTED1.START.CHAR = DATA2
      TEXTED1.START.LINE = INT(DATA1 / 65536) ❶
      TEXTED1.END.CHAR = DATA4
      TEXTED1.END.LINE = INT(DATA3 / 65536) ❶

      IF TEXTED1.END.CHAR = 0 AND TEXTED1.END.LINE = 0 THEN ❷
         CALL Disable(CONTEXT, EditCut, ERR)
         CALL Disable(CONTEXT, EditCopy, ERR)
      ELSE
         CALL Enable(CONTEXT, EditCut, ERR)
         CALL Enable(CONTEXT, EditCopy, ERR)
      END
   END CASE
```

❶ The values returned in the DATA1 and DATA3 parameters are offset by 65536, and must be adjusted to obtain the true value.

❷ If there is no selection, all four character-position parameters are zero. In practice, however, only the end-position parameters need be tested.

Notes:

1. Since it is unlikely that any text will be selected when your application starts, you should disable the Cut and Copy menu items in your resource script.

2. Text in an edit box can also be selected by calling the **EditBoxSetSelected** subroutine. This does not, however, generate a Select message. If you use this subroutine, you should enable or disable the Cut and Copy menu items as appropriate.

The Paste Item

You will need to enable the Paste menu item when there is data available on the clipboard, and disable it when there is none. Within your application, this can be done whenever you call the **ClipboardSetContent**, **Cut** or **Copy** subroutines. However, other applications can also place data on the clipboard and your application must test the contents of the clipboard whenever another application has had control.

The easiest way to do this is to process the **UIMS.MSG.ENTER** message. Change your message loop to include the following:

```
USER.WANTS.TO.EXIT = FALSE
LOOP UNTIL USER.WANTS.TO.EXIT DO

   CALL GetMsg(0, ...
               MSG.CONTEXT, ...
               MSG.WINDOW, ...
               MSG.CONTACT, ...
               MSG.TYPE, ...
               TIMESTAMP, ...
               DATA1, ...
               DATA2, ...
               DATA3, ...
               DATA4)

   BEGIN CASE ❶
   CASE MSG.TYPE = UIMS.MSG.ENTER
      CALL ClipboardGetSize("TEXT", CLIP.SIZE) ❷
      IF CLIP.SIZE = 0 THEN
         CALL Disable(CONTEXT, EditPaste, ERR)
      ELSE
         CALL Enable(CONTEXT, EditPaste, ERR)
      END
   END CASE

   BEGIN CASE
   CASE MSG.WINDOW = Win1
         .
         . REM processing for Win 1
         .
   END CASE
   .
   . REM processing for other windows
   .
REPEAT
```

❶ Because any contact might be selected when the application receives the focus, all Enter messages must be processed. This is done by processing them outside the case statements for the individual windows.

❷ The amount of text on the clipboard is tested by calling **ClipboardGetSize**. If there is text available, the Paste menu item is enabled; otherwise it is disabled.

An Example Application: ClipEdit

This example application illustrates how to cut and copy to, and paste from the clipboard. To create the ClipEdit application, copy and rename the source files for the EditCtrl application, and then make the following changes:

1. In the resource script, set initial states for the Cut, Copy, Paste and Clear menu items, and the Cut, Copy and Paste buttons.
2. Enable select messages.
3. Add new variables.
4. Modify the UIMS.MSG.ENTER case to test whether there is any text on the clipboard.
5. Add a UIMS.MSG.SELECT case to the message loop.
6. Modify the HANDLE.WIN1.MENUS subroutine to process the Cut, Copy, Paste and Clear commands.
7. Modify the HANDLE.WIN1.BUTTONS subroutine to process the Cut, Copy and Paste button operations.
8. Add subroutines to support the Edit commands.
9. Compile the resource file and the DATA/BASIC program.

Modify the Resource Script

When you start the ClipEdit application, no text will be selected; the Cut, Copy and Clear commands on the Edit menu must therefore be disabled. The Paste menu item can also be disabled, though its state will in fact be set within the DATA/BASIC program.

Edit the resource script, and add plus signs to the titles of the Cut, Copy, Paste and Clear items on the Edit menu as shown below. At the same time, you can remove the plus sign from the Undo menu item.

```
MENU = 0
{
  TITLE = '&Edit'
  CHILDREN = '&Undo'      = EditUndo,
             '-'           = 0,
             'Cu&t+'       = EditCut,      /* disabled */
             '&Copy+'      = EditCopy,     /* disabled */
             '&Paste+'     = EditPaste,   /* disabled */
             'C&lear+'      = EditClear   /* disabled */
}
```

Since the Cut, Copy and Paste menu items will be disabled, the corresponding buttons must also be disabled. Find the definitions for the Cut, Copy and Paste buttons and add the following line to each:

```
ENABLED = FALSE
```

Enable Select Messages

Your application will need to monitor Select messages in order to keep track of any text selection within the TextEditor. Find the following line in your DATA/BASIC source:

```
CALL SetEventMask(CONTEXT, CONTEXT, EVENTMASK, ERR)
```

and add the following lines just before it:

```
CALL BitTest(EVENTMASK, UIMS.EM.SELECT, ENABLED)
IF NOT(ENABLED) THEN EVENTMASK = EVENTMASK + UIMS.EM.SELECT
```

Add New Variables

You will need four variables to store the start and finish positions of any text selection. Add the following lines to your DATA/BASIC source anywhere before the start of the message loop:

```
* Initialise the text selection variables
TEXTED1.START.CHAR = 0
TEXTED1.START.LINE = 0
TEXTED1.END.CHAR = 0
TEXTED1.END.LINE = 0
```

Modify the Enter Case

The Paste commands (that is, the Paste button and the Paste item on the Edit menu) will be enabled and disabled depending on whether or not there is any text on the clipboard. The application must call **ClipboardGetSize** to test the contents of the clipboard and then use **SetEnabled** (or **Enable** and **Disable**) to set the relevant contacts to the appropriate state.

In the ClipEdit application, this will be done whenever a **UIMS.MSG.ENTER** message is received. Find the following case statement in the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.ENTER
```

and add the following just after it:

```
* Find out if there is any text on the clipboard and
* enable or disable the Paste commands as appropriate
CALL ClipboardGetSize("TEXT", CLIP.SIZE)
CALL SetEnabled(CONTEXT, EditPaste, (CLIP.SIZE # 0), ERR)
CALL SetEnabled(CONTEXT, PasteButton, (CLIP.SIZE # 0), ERR)
```

Add the Select Case

The Cut, Copy and Clear commands must be enabled when text is selected, and disabled otherwise. This is done by monitoring select messages. Add the following to the case statement in the HANDLE.WIN1.MESSAGES subroutine:

```
CASE MSG.TYPE = UIMS.MSG.SELECT
  BEGIN CASE
  CASE MSG.CONTACT = TxtEd1
    TEXTED1.START.CHAR = DATA2
    TEXTED1.START.LINE = INT(DATA1 / 65536) ❶
    TEXTED1.END.CHAR = DATA4
    TEXTED1.END.LINE = INT(DATA3 / 65536) ❶

    SELECTED = (TEXTED1.END.CHAR # 0) OR (TEXTED1.END.LINE # 0) ❷
    CALL SetEnabled(CONTEXT, EditCut, SELECTED, ERR)
    CALL SetEnabled(CONTEXT, EditCopy, SELECTED, ERR)
    CALL SetEnabled(CONTEXT, EditClear, SELECTED, ERR)
    CALL SetEnabled(CONTEXT, CutButton, SELECTED, ERR)
    CALL SetEnabled(CONTEXT, CopyButton, SELECTED, ERR)
  END CASE
```

❶ The line numbers returned in the DATA1 and DATA3 variables are offset by 65536 and must be divided by this value to obtain the true line numbers.

❷ This line sets the SELECTED variable to TRUE (1) if both finish-position parameters are non-zero (text selected), or to FALSE (0) if they are both zero (no text selected). The resulting value is used in the calls to **SetEnabled** to set the Cut, Copy and Clear commands to the appropriate state.

Modify the Edit Menu Commands

In the EditCtrl application, all the Edit menu commands display the message 'Command not implemented!'. In ClipEdit, these commands must be changed to carry out the appropriate operations.

Find the Edit menu commands in the HANDLE.WIN1.MENU subroutine, and replace them with the following:

```
* Edit menu commands
CASE MSG.CONTACT = EditUndo

  CALL CreateMessageBox(CONTEXT, ...
                        UIMS.INFO, ...
                        "ClipEdit Example Application", ...
                        "Command not implemented!", ...
                        "", ...
                        OK, ...
                        ERR) ❶

CASE MSG.CONTACT = EditCut
  CALL Cut(CONTEXT, TxtEd1, -1, -1, -1, -1, ERR) ❷
  CALL Enable(CONTEXT, EditPaste, ERR) ❸
  CALL Enable(CONTEXT, PasteButton, ERR) ❸

CASE MSG.CONTACT = EditCopy
  CALL Copy(CONTEXT, TxtEd1, -1, -1, -1, -1, ERR) ❷
  CALL Enable(CONTEXT, EditPaste, ERR) ❸
  CALL Enable(CONTEXT, PasteButton, ERR) ❸

CASE MSG.CONTACT = EditPaste
  GOSUB EDIT.PASTE

CASE MSG.CONTACT = EditClear
  GOSUB EDIT.CLEAR
```

❶ The Undo command remains unimplemented, so an appropriate message is displayed when this command is selected.

❷ The Cut and Copy commands will operate on the currently selected text, so the start- and finish-position parameters are all set to -1.

❸ The Cut and Copy commands place text on the clipboard, so the Paste commands must be enabled.

Modify the Edit Button Actions

In the EditCtrl application, all the Edit buttons display the message 'Command not implemented!'. In ClipEdit, these commands must be changed to carry out the appropriate operations.

Find the Edit button commands in the HANDLE.WIN1.BUTTONS subroutine, and replace them with the following:

```
* Edit actions
CASE MSG.CONTACT = CutButton
  CALL Cut(CONTEXT, TxtEd1, -1, -1, -1, -1, ERR)
  CALL Enable(CONTEXT, EditPaste, ERR)
  CALL Enable(CONTEXT, PasteButton, ERR)

CASE MSG.CONTACT = CopyButton
  CALL Copy(CONTEXT, TxtEd1, -1, -1, -1, -1, ERR)
  CALL Enable(CONTEXT, EditPaste, ERR)
  CALL Enable(CONTEXT, PasteButton, ERR)

CASE MSG.CONTACT = PasteButton
  GOSUB EDIT.PASTE

CASE MSG.CONTACT = UndoButton
  CALL CreateMessageBox(CONTEXT, ...
                        UIMS.INFO, ...
                        "ClipEdit Example Application", ...
                        "Command not implemented!", ...
                        "", ...
                        OK, ...
                        ERR)
```

Each of these commands is a duplicate of the corresponding command on the Edit menu.

Add Support Subroutines

You need to add subroutines to your DATA/BASIC source file to support the Paste and Clear commands. These are:

EDIT.PASTE	This pastes text from the clipboard into the TextEditor.
EDIT.CLEAR	This deletes the selected text from the TextEditor. This subroutine is called when the Clear command is selected.

EDIT.PASTE Subroutine

This subroutine pastes the text from the clipboard at the current cursor position, by calling the **Paste** subroutine with both position parameters set to -1. If any text was selected, this will be replaced and the cursor will be positioned at the end of the new text, with no text selected; the Cut, Copy and Clear commands must therefore be disabled. Finally, the TEXTED1.END.CHAR and TEXTED1.END.LINE variables are set to zero to show that no text is selected.

```
***************************************************************
*
* SUBROUTINE: EDIT.PASTE
*
* PURPOSE: Pastes text from the clipboard into the TextEditor
*
* COMMENTS:
*   After pasting no text will be selected, so the Cut, Copy
*   and Clear commands must be disabled.
*
***************************************************************

  * Paste at the current cursor position
  CALL Paste(CONTEXT, TxtEd1, -1, -1, ERR)

  * No text is now selected, so disable the Cut, Copy and Clear
  * commands
  CALL Disable(CONTEXT, EditCut, ERR)
  CALL Disable(CONTEXT, EditCopy, ERR)
  CALL Disable(CONTEXT, EditClear, ERR)
  CALL Disable(CONTEXT, CutButton, ERR)
  CALL Disable(CONTEXT, CopyButton, ERR)

  * Cursor is now an insertion point
  TEXTED1.END.CHAR = 0
  TEXTED1.END.LINE = 0

RETURN
```

EDIT.CLEAR Subroutine

This subroutine fetches the contents of the TextEditor by calling the **TextEditorGetContent** subroutine. It then uses the position parameters set by the last select message to calculate the character positions for the start and finish of the text to be deleted. Next, it constructs a new text string by concatenating the text from the beginning of the string to the start of the deleted section, with the text from the end of the deleted section to the end of the string. Finally, **TextEditorSetContent** is called to insert the new string into the TextEditor, replacing the previous contents.

```
***************************************************************
*
* SUBROUTINE: EDIT.CLEAR
*
* PURPOSE: Deletes text without copying it to the Clipboard
*
* COMMENTS:
*   On exit, the cursor will be positioned at the start of the
*   text.
*
***************************************************************

EDIT.CLEAR:
  * Get the contents of the edit control.
  CALL TextEditorGetContent(CONTEXT, TxtEd1, TEXT, ERR)

  * Delete the currently selected text.
  * The selection details are provided by the latest Select message.
  STARTPOS = INDEX(TEXT, CHAR(254), TEXTED1.START.LINE) + ...
               TEXTED1.START.CHAR
  ENDPOS = INDEX(TEXT, CHAR(254), TEXTED1.END.LINE) + ...
             TEXTED1.END.CHAR - 1
  TEXT = TEXT[1, STARTPOS]:TEXT[ENDPOS+2, LEN(TEXT)-ENDPOS]

  * Restore the contents of the edit control.
  CALL TextEditorSetContent(CONTEXT, TxtEd1, TEXT, ERR)

RETURN
```

Compile

When you have made these changes, you can compile the resource script and DATA/BASIC program as described in Chapter 4 for the Generic application. When you run ClipEdit, you will be able to use the Cut, Copy, Paste and Clear commands to edit text within the TextEditor control. Note that the Cut, Copy and Clear commands are only enabled if text is selected, and that the Paste command is only enabled if there is text on the clipboard.

Chapter 12
Fonts

UIMS allows you to improve the look of your applications by using different fonts. A font is a collection of characters that have a unique combination of height, width, typeface, character set, and other attributes. An application uses fonts to display text in various typefaces and sizes. For example, a word processor uses fonts to give a 'what-you-see-is-what-you-get' interface.

This chapter describes how to use fonts in your application and explains how to create an example application that illustrates these concepts.

Writing Text

There are two ways of writing text in a window:

- You can write text directly onto the client area, or the text canvas if the window has one, by using the **DrawTextString** subroutine.
- You can create a **Text** contact containing the text you wish to display.

In the first case, the font used will be that attached to the window's Drawrule, whereas a **Text** contact can have its own Drawrule, and thus its own font.

The default font is the system font, a variable width font representing characters in the ANSI character set. The typeface used for the system font is called 'System'. UIMS uses the system font for menus, window titles and other text.

Using Colour when Writing Text

You can use any colour for the text you write by setting the foreground and background colours for the window's or text object's Drawrule. The foreground colour determines the colour of the character to be written, while the background colour determines the colour of everything else in the character cell (the rectangle enclosing the character). A character cell usually has the same width and height as the character.

The foreground and background colours can be set when the Drawrule is created, or by calling the **DrawruleSetColour** subroutine. The following example sets the foreground colour to red and the background to green:

```
CALL DrawruleSetColour(CONTEXT, ...
                       Drawrule1, ...
                       UIMS.RED, ...
                       UIMS.GREEN, ...
                       ERR)
```

The background colour is only used if the drawrule's text mode is set to opaque. The text mode determines whether or not the background colour in the character cell effects what is already on the display. If the mode is opaque, the background colour overwrites anything already on the display; if it is hollow, anything on the display that would otherwise be overwritten by the background is preserved. You set the text mode when you create your Drawrule.

Note: Although you can write text on the client area using different colours, if you have a text canvas, the text will always be redrawn in a single colour. Refer to page 12-7 for details of how to use multiple fonts and colours.

Creating a Font

If you want to use more than one font in your application, you will need to create a Font object for each combination of typeface, style and point size. You create a new Font object with the **CreateDrawFont** subroutine, specifying the typeface, style and point size required. Once created a Font object must be attached to a Drawrule before it can be used.

The following example creates a bold 10pt Times font:

```
CALL CreateDrawFont(CONTEXT, ...
                    TimesFont, ...
                    UIMS.FONT.BOLD, ...
                    Times❶, ...
                    10, ...
                    TIMES.FONT)

CALL DrawruleSetFont(CONTEXT, Drawrule1, TimesFont, ERR)
```

❶ This parameter must be the handle of a TypeFace object. Refer to the next section for how to find out which typefaces are available.

TypeFaces

UIMS fonts can only use the typefaces that are available on Windows. You can find out which typefaces are available by using the **GetTypeFaces** subroutine to obtain a list of handles to these typefaces. You can use the **TypeFaceGetName** subroutine to find out the name of a particular typeface.

For example, the following finds out whether the Helvetica typeface is available on your PC:

```
CALL GetTypeFaces(TYPEFACES❶, ERR)
NAME = ""
FOR I = 1 TO DCOUNT(TYPEFACES, CHAR(254)) WHILE NAME <> "Helvetica"
  HANDLE = TYPEFACES<I>
  CALL TypeFaceGetName(CONTEXT, HANDLE, NAME, ERR)
NEXT I
IF NAME <> "Helvetica" THEN AVAIL = "Helvetica is not available"...
ELSE AVAIL = "Helvetica is available"
CALL DrawTextString(CONTEXT, Win1, AVAIL, 10, 10, ERR)
```

❶ The list of typeface handles is returned as a dynamic array, with one handle to each attribute.

Point Sizes

When setting the point size for a font, you should ideally use a size that is available in the selected typeface. You can use the **TypeFaceGetPointSizes** subroutine to obtain a list of point sizes for a specific typeface. Note, however, that you can generally choose any point size – if the required size is not available, UIMS will try to create it by scaling one of the available sizes; if this cannot easily be done, the nearest equivalent will be selected.

Note: Some typefaces (for example, the TrueType fonts supplied with Windows 3.1) can be scaled to any size. In this case, **TypeFaceGetPointSizes** returns only a single size.

Type Styles

Most typefaces are also available in a range of styles. UIMS allows you to choose any combination of normal, bold, italic, underlined, strikeout and outline. Where a particular style is available for the typeface concerned, UIMS will use it; otherwise UIMS will try to synthesise the style. If the style cannot be easily be synthesised, the nearest equivalent will be selected.

Note: Some styles are particularly difficult to synthesise. In particular, outline cannot generally be used unless the typeface concerned includes an outline style. Similarly, for some typefaces, it may not be possible to use strikeout style.

Getting Information about a Font

Once you have created a Font object, you can obtain the handle of its **TypeFace** by calling the **FontGetTypeFace** subroutine. You can also find out the current point size and style with **FontGetPointSize** and **FontGetStyle** respectively.

Two other subroutines, **FontGetMetrics** and **FontGetTextLen**, give you details of your particular combination of typeface, point size and style. **FontGetMetrics** returns the dimensions of the font in pixels – the following example obtains the dimensions of the font with the identifier Font1:

```
CALL FontGetMetrics(CONTEXT, ...
                    Font1, ...
                    HEIGHT❶, ...
                    ASCENT❷, ...
                    DESCENT❸, ...
                    LEADING❹, ...
                    LCWIDTH❺, ...
                    UCWIDTH❻, ...
                    MAXWIDTH❼, ...
                    ERR)
```

The parameters (with the exception of the first two and the last) must be variables in which to return the following information.

❶ The total height of the font – the ascent plus the descent (see below).

❷ The ascent – the height above the base line of the tallest characters.

❸ The descent – the height of the longest descender.

❹ The leading – the distance between the descenders of one row of characters and the top of the tallest characters in the next row.

❺ The average width of the lower case characters in a proportionally spaced font. For fixed pitch fonts this is the width of a single character.

❻ The average width of the upper case characters in a proportionally spaced font. For fixed pitch fonts this is the width of a single character.

❼ The width of the widest character.

These dimensions are illustrated in Figure 12-1.

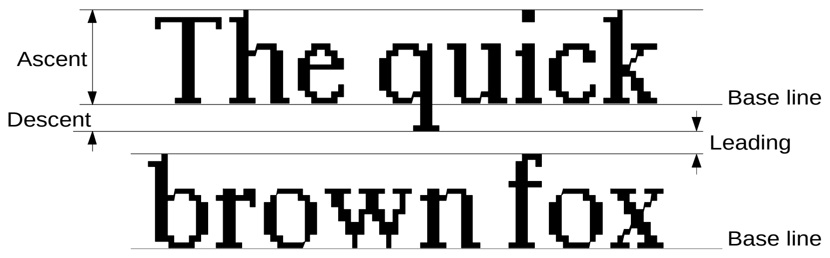


Figure 12-1. Font Metrics

In proportionally spaced fonts, each character is a different width. This means that, although **FontGetMetrics** can tell you the average widths of the lower- and upper-case characters in the font, different strings will be of different lengths when displayed, even though they might have the same number of characters. The **FontGetTextLen** subroutine returns the length in pixels of a specified string, when displayed using a specified font object. For example, the following obtains the length of the string "Hello, world!" when displayed in the font identified by Font1:

```
CALL FontGetTextLen(CONTEXT, Font1, "Hello, world!", LENGTH)
```

The length is returned in the final parameter.

Using Multiple Fonts in a Window

If you are developing an application that uses a variety of fonts, you may need to use more than one font in the same window. If you have a number of independent text items, each in a different font, your task is easy – simply make each item a different Text contact, or create a child window to hold the text. If, however, you want to use more than one font in the same line of text, your task is more difficult.

There are two ways of approaching this problem: you can monitor update messages and redraw the window when necessary, or you can give your window a text canvas for the bulk of the text and use text contacts for the items that are in a different font. In either case, you will need to keep track of the length of the line as each text item is displayed, so that you know where to start the next part of the line. If you are using a proportionally spaced typeface, each character has a different width, making it difficult to calculate the length of a string. However, UIMS provides the **FontGetTextLen** subroutine, which computes the length of a given string when printed in a specified font.

Redrawing when Necessary

One way to draw a line of text that contains multiple fonts is to use **FontGetTextLen** after each call to **DrawTextString**, and add the length to the current position. The following example shows how to write the string "This is an example string.", using italic characters for the word "example", and normal characters for all the others:

```
CALL GetDrawrule(CONTEXT, Win1, WIN1.DRAWRULE)
CALL DrawruleGetFont(CONTEXT, WIN1.DRAWRULE, WIN1.FONT)
.
.
.
X = 10   ;* horizontal start point

CALL FontSetStyle(CONTEXT, WIN1.FONT, UIMS.NONE, ERR)
TEXT = "This is an "
CALL DrawTextString(CONTEXT, Win1, TEXT, X, 10, ERR)
CALL FontGetTextLen(CONTEXT, WIN1.FONT, TEXT, LENGTH)
X = X + LENGTH   ;* update the print position

CALL FontSetStyle(CONTEXT, WIN1.FONT, UIMS.FONT.ITALIC, ERR)
TEXT = "example"
CALL DrawTextString(CONTEXT, Win1, TEXT, X, 10, ERR)
CALL FontGetTextLen(CONTEXT, WIN1.FONT, TEXT, LENGTH)
X = X + LENGTH   ;* update the print position

CALL FontSetStyle(CONTEXT, WIN1.FONT, UIMS.NONE, ERR)
```

```
TEXT = " string"
CALL DrawTextString(CONTEXT, Win1, TEXT, X, 10, ERR)
```

In this example, the line is divided in to sections and for each, **FontSetStyle** sets normal or italic style, and **DrawTextString** prints the text. The length of the printed text is then calculated and added to the current position to set the starting location for the next section.

As an alternative to using **FontSetStyle**, you could create an italic Font with **CreateDrawFont**, and use **DrawruleSetFont** to change fonts as required, or create both a Font and a Drawrule, and use **SetDrawrule** to change Drawrules.

Using a Text Contact

You would use the method shown above to redraw the window each time your application receives an update message. Alternatively, you could use a text canvas to manage the normal text, and create a Text contact for the italic text. As before, you must position the normal text to allow for that in italics:

```
X = 10   ;* horizontal start point

TEXT = "This is an "
CALL DrawTextString(CONTEXT, Win1, TEXT, X, 10, ERR)
CALL FontGetTextLen(CONTEXT, NormalFont, TEXT, LENGTH)
X = X + LENGTH   ;* update the print position

TEXT = "example"
CALL CreateText(CONTEXT, ...
                ItalicText, ...
                TEXT, ...
                X, 10, ... ❶
                0, 0, ... ❷
                "", ...
                ITALIC.TEXT)
CALL SetDrawrule(CONTEXT, ItalicText, ItalicDrawrule, ERR) ❸
CALL AddChild(CONTEXT, Win1, -1, ItalicText, ERR) ❹

CALL FontGetTextLen(CONTEXT, ItalicFont, TEXT, LENGTH)
X = X + LENGTH   ;* update the print position

TEXT = " string"
CALL DrawTextString(CONTEXT, Win1, TEXT, X, 10, ERR)
```

❶ These two parameters specify the position of the Text contact. The horizontal position is set as in the previous example; that is by adding the length of the preceding text to a known position.

❷ These two parameters specify the size of the Text contact. A width and height of zero instruct the contact to set its size to that of its contents; that is, the text "example". Note, however, that the size will not be calculated until the contact is given a parent, and this allows the Font to be changed first.

❸ This line sets the Drawrule (and thus the italic Font) for the Text contact.

❹ This line makes the Text contact a child of Win1, thus displaying it in the window.

It is assumed that the necessary Drawrules and Fonts will have been previously created.

Printing in Different Colours

The techniques described above can also be used if you want to use different colours for different parts of a line of text. For example, to change colours while redrawing the client area, you would use the **DrawruleSetColour** subroutine to change colours as required.

An Example Application: ShowFont

The example application, Showfont, illustrates how to use different typefaces, and demonstrates the two methods discussed above for using more than one font in a line of text.

Showfont is an extension of the Output application described in Chapter 5. To create the Showfont application, copy and rename the source files of the Output application and then make the following modifications:

1. Add new constant definitions.
2. Define a Drawrule resource.
3. Modify the UIMS.MSG.CREATE case.
4. Modify the UIMS.MSG.UPDATE case.
5. Compile the resource file and the DATA/BASIC program.

Add New Constant Definitions

You will need identifiers for the additional resources defined in the resource script. These must be available to both the resource script and the DATA/BASIC source, so add the following to your header file:

```
EQUATE ItalicText TO 200
EQUATE Drawrule2 TO 210
EQUATE ItalicFont TO 211
```

Ensure that the new header file is available on both the host and the PC.

Define New Resources

You will need an additional Drawrule for the italic font and this can be created in the resource script. Add the following lines to the file SHOWFONT.UCL:

```
DRAWRULE = Drawrule2
{
  FOREGROUND = BLACK
  BACKGROUND = WHITE
  DRAWMODE = COPY
  TEXTMODE = OPAQUE
}
```

Note that Font objects cannot be created in the resource script, so the italic font must be created in the application by calling **CreateDrawFont**.

Modify the Create Case

In the Output application, the **UIMS.MSG.CREATE** message is used to draw text and display objects in the left-hand Child window, Child1. In Showfont, it will also create the italic font, and create and position a text object.

In the HANDLE.WIN1.CREATE subroutine in your DATA/BASIC source, find the lines which read:

```
CALL GetDrawrule(CONTEXT, Child1, DEFAULT.DRAWRULE)
CALL DrawruleGetFont(CONTEXT, DEFAULT.DRAWRULE, DEFAULT.FONT)
```

and immediately after them, add the following.

```
* Change the default font to 12pt Times normal
CALL GetTypeFaces(TYPEFACES, ERR)
NAME = ""
FOR I = 1 TO DCOUNT(TYPEFACES, CHAR(254)) UNTIL NAME[1, 5] = "Times"
  CALL TypeFaceGetName(CONTEXT, TYPEFACES<I>, NAME, ERR)
NEXT I
* If Times not available, use the system typeface
IF I > DCOUNT(TYPEFACES, CHAR(254)) THEN
  CALL GetDefaults(P1, P2, TIMES.TFACE, ERR)
END ELSE
  TIMES.TFACE = TYPEFACES<I-1>
END
CALL FontSetTypeFace(CONTEXT, DEFAULT.FONT, TIMES.TFACE, ERR)
CALL FontSetPointSize(CONTEXT, DEFAULT.FONT, 12, ERR)
CALL FontSetStyle(CONTEXT, DEFAULT.FONT, UIMS.NONE, ERR)

* Create the italic font
CALL CreateDrawFont(CONTEXT, ...
                    ItalicFont, ...
                    UIMS.FONT.ITALIC, ...
                    TIMES.TFACE, ...
                    12, ...
                    ITALIC.FONT)
* Attach it to Drawrule 2, which was loaded from the resource file
CALL DrawruleSetFont(CONTEXT, Drawrule2, ItalicFont, ERR)
```

Then find the lines which read:

```
TEXT = "Canvas and UIMS therefore manages"
CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)
VPOS = VPOS + VSPACE
```

and replace them with:

```
* The next line will include text in the italic font
* Print the first part
TEXT = "Canvas and "
CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)

* Set the horizontal position for the italic text
CALL FontGetTextLen(CONTEXT, DEFAULT.FONT, TEXT, LENGTH)
HPOS = HPOS + LENGTH

* Create a Text contact without a parent
TEXT = "UIMS"
CALL CreateText(CONTEXT, ...
                ItalicText, ...
                TEXT, ...
                HPOS, VPOS, ...
                0, 0, ...
                "", ...
                ITALIC.TEXT)

* Set its Drawrule to that with the italic font
CALL SetDrawrule(CONTEXT, ItalicText, Drawrule2, ERR)

* Attach it to the child window
CALL AddChild(CONTEXT, Child1, -1, ItalicText, ERR)

* Calculate the horizontal position for the rest of the line
CALL FontGetTextLen(CONTEXT, ItalicFont, TEXT, LENGTH)
HPOS = HPOS + LENGTH

* Print the rest of the line
TEXT = " therefore manages"
CALL DrawTextString(CONTEXT, Child1, TEXT, HPOS, VPOS, ERR)

* Set the drawing position for the next line
VPOS = VPOS + VSPACE
HPOS = UCWIDTH
```

Modify the Update Case

In the Output application, the **UIMS.MSG.UPDATE** message is used to draw text and graphics in the right-hand Child window, Child2. In Showfont, this is modified to draw some of the text in the italic font.

In the subroutine HANDLE.CHILD2.MESSAGES find the lines which read:

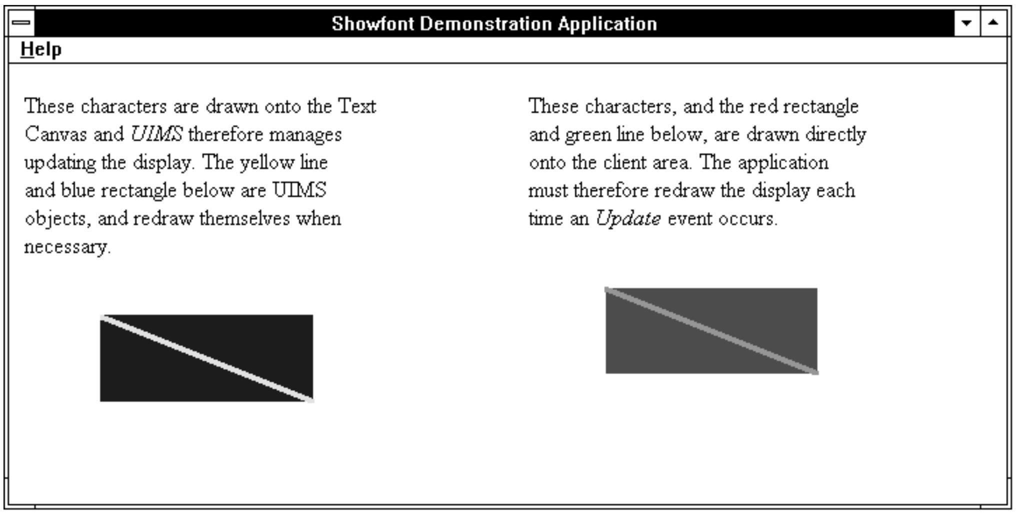
```
TEXT = "time an Update message is received."
CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)
VPOS = VPOS + VSPACE
```

and replace them with:

```
* The next line will include text in the italic font
* Print the first part
TEXT = "time an "
CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)

* Set the horizontal position for the italic text
CALL FontGetTextLen(CONTEXT, DEFAULT.FONT, TEXT, LENGTH)
HPOS = HPOS + LENGTH

* Change to the italic style
CALL FontSetStyle(CONTEXT, DEFAULT.FONT, UIMS.FONT.ITALIC, ERR)

* Print the italic text
TEXT = "Update"
CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)

* Calculate the horizontal position for the rest of the line
CALL FontGetTextLen(CONTEXT, DEFAULT.FONT, TEXT, LENGTH)
HPOS = HPOS + LENGTH

* Restore the original style
CALL FontSetStyle(CONTEXT, DEFAULT.FONT, UIMS.NONE, ERR)

* Print the rest of the line
TEXT = " message is received."
CALL DrawTextString(CONTEXT, Child2, TEXT, HPOS, VPOS, ERR)

* Set the drawing position for the next line
HPOS = UCWIDTH
VPOS = VPOS + VSPACE
```

Compile

When you have made these changes, you can compile the resource script and DATA/BASIC program as described in Chapter 4 for the Generic application. When run, the application should look like this:



These characters are drawn onto the Text Canvas and *UIMS* therefore manages updating the display. The yellow line and blue rectangle below are UIMS objects, and redraw themselves when necessary.

These characters, and the red rectangle and green line below, are drawn directly onto the client area. The application must therefore redraw the display each time an *Update* event occurs.

Figure 12-2. The Showfont Application

Chapter 13
Dynamic Data Exchange

UIMS provides two ways of communicating and sharing information with other Windows applications. Chapter 11 describes how to use the Cut, Copy and Paste subroutines to transfer data via the Windows Clipboard. However, a more flexible method is to use Dynamic Data Exchange (DDE). This allows you to extract information from other Windows applications, to automatically update them with new information, and even to send commands to manipulate them by remote control.

Introduction

Dynamic Data Exchange (DDE) is a mechanism provided by Microsoft Windows that enables two Windows applications to communicate with each other. DDE automates the manual cutting and pasting of information between applications, thus making manual intervention unnecessary.

DDE can be used in four ways:

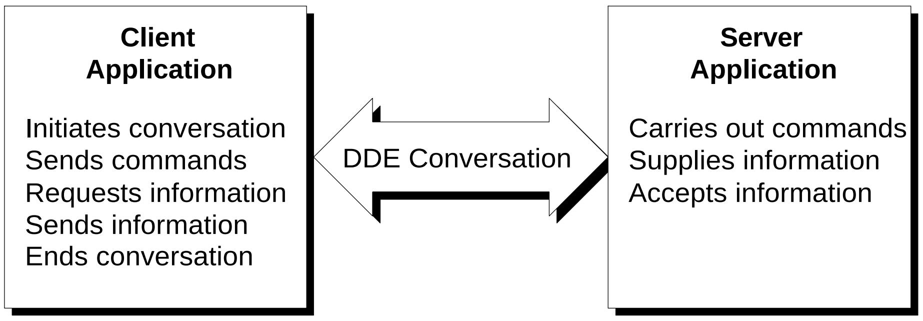
- You can request information from an application. For example, in a DDE conversation with Microsoft Word for Windows, a UIMS application can request the contents of part or all of a Word document.
- You can send information to an application. For example, in a DDE conversation with Microsoft Word, a UIMS application can send text to a specified location in a document.
- You can send commands to an application. For example, in a DDE conversation with Microsoft Word, a UIMS application can send a command to open a document from which it requires information. Commands sent to an application must be in a form the application can recognise.
- An 'advise' link can be established, over which an application will send updated information each time a change occurs.

Note: Not all Windows applications support DDE. Consult the documentation for your Windows applications to see if they support DDE.

An Overview of DDE

Clients, Servers and Conversations

Two applications exchange information by engaging in a DDE *conversation*. In a DDE conversation, the application that initiates and controls the conversation is called the *client* and the application that responds is the *server*. The client application requests information from the server and sends information and commands to it. The server application, as its name implies, serves the needs of the client application by returning information, accepting information and carrying out commands. There is nothing special about an application that makes it a client or a server; they are simply the roles that an application can adopt. In fact, an application can be engaged in several DDE conversations at the same time, acting as the client in some and the server in others.



This chapter describes how to use your UIMS application as a DDE client. UIMS applications cannot be used as servers. However, provided it is not running a UIMS application, RealLink for Windows can be used as a DDE server – refer to the *RealLink for Windows User Manual* for details.

Note: Some applications and programming languages refer to the DDE client as the *destination*, and to the server as the *source*.

Applications, Topics and Items

When a client application initiates a DDE conversation, it must specify two things:

- The name of the application with which it wishes to communicate (the server).
- The subject of the conversation (the *topic*).

When a server application receives a request for a conversation about a topic that it recognises, it responds and the conversation commences. Once established, a conversation cannot change to a different application or topic. If communication is required with another application or on a different topic, a new conversation must be started.

During the conversation, the client and server can exchange information about one or more *items*. An item is a reference to data that is meaningful to the server application. Both the client and the server can change the item during the conversation.

Applications

Every Windows application that supports DDE has a unique DDE application name. This is usually, but not always, the name of the executable file for that application, without the .EXE filename extension. For example, The DDE application name for Microsoft Excel is Excel, and that for Word for Windows is WinWord. However, the DDE application name for RealLink for Windows is RFWDDE, not RFW. If you are not sure what an application's DDE application name is, look in the documentation for that application.

Note: DDE application names are not case sensitive.

Topics

Every DDE conversation is on a topic that the server application recognises. Most applications support the names of open files as topics. Some possible topics are a Microsoft Excel worksheet (for example, ACCOUNTS.XLS), a Word for Windows document (for example, MEMO.DOC), or a RealLink for Windows configuration file (for example, STAFF.CFG).

A special topic that many applications recognise is "System". This provides a list of the other topics that are currently available and other information about the application. Unlike other topics, which may or may not be available, depending on whether a file is open, the System topic is always available.

Items

Given the server's application name and a topic name, a client can initiate a DDE conversation. However, for a client to exchange information with a server, one other essential piece of information is needed: the items available in the topic of the DDE conversation. An item is a sub-topic that identifies the information actually being exchanged during the DDE conversation. For example, Microsoft Excel recognises cell references (such as R1C1) as items in a conversation. In Word for Windows, a bookmark is an item, while in RealLink for Windows, the Cursor, Row, Table and RxBuff keywords are items. Refer to the documentation for your application to find out what items are recognised.

'Advise' Data Links

An advise data link is one in which the server notifies the client of any changes to a given data item. This notification process continues until the DDE conversation is terminated.

Windows provides two kinds of advise DDE link: 'hot' links, in which the server immediately sends the changed data to the client; and 'warm' links, in which the server advises the client that the data has changed, but does not actually send the data until the client requests it. The UIMS DDE mechanism is similar to a warm link, in that the server application must poll the link to find out when updated data is available.

Using the UIMS DDE Subroutines

The UIMS DATA/BASIC API provides six DDE commands:

DDE.PEEK	Requests data from a server application.
DDE.POKE	Sends data to a server application.
DDE.EXECUTE	Sends one or more commands to a server application.
DDE.OPENADVISE	Establishes an advise DDE link to a Windows application.
DDE.ADVISE	Obtains data from an advise DDE link.
DDE.CLOSEADVISE	Closes an advise DDE link.

Each of the first three subroutines initiates a conversation with the specified application, carries out the appropriate action, and then terminates the conversation. There are no separate subroutines that initiate and terminate DDE conversations. As a result, the each conversation is uni-directional and passes information about a single item only.

The other three subroutines must be used in combination. Call **DDE.OPENADVISE** to create the link, and then periodically call **DDE.ADVISE** to find out whether the data has changed and obtain any changes. When the link is no longer required, terminate it with **DDE.CLOSEADVISE**.

Requesting Information

To request information from a DDE server, you use the **DDE.PEEK** subroutine. You must provide the name of the application, a topic recognised by that application and an item within that topic. You must also provide two variables in which the requested information and the completion status of the subroutine can be returned. For example, if you need to ask Microsoft Excel for a list of the currently supported topics, you would use the following DATA/BASIC statement.

```
CALL DDE.PEEK("Excel"❶, "System"❷, "Topics"❸, TOPICS❹, ERR❺)
```

❶ This is Microsoft Excel's DDE application name.

❷ This parameter specifies the System topic.

❸ "Topics" is the name of an item that Excel provides in the System topic. It lists all the topics that are currently available.

❹ This parameter is a variable in which the requested data will be returned.

❺ This parameter is a variable in which the completion status of the subroutine will be returned.

The topic names returned are separated by tabs. If you want to display the list in a message box, you should convert these to line feeds as follows:

```
TOPICS = CHANGE(TOPICS, CHAR(9), CHAR(10))
CALL CreateMessageBox(CONTEXT, ...
                      UIMS.INFO, ...
                      "Microsoft Excel - DDE Topics", ...
                      TOPICS, ...
                      "", ...
                      OK, ...
                      ERR)
```

Similarly, if you want to display the list in a list box, you should convert the tabs to attribute marks.

Note: If the requested item is not recognised by the server application, an empty string is returned.

Sending Information

Although the client in a DDE conversation usually requests information from the server, the client can also send information to the server. To do this, you use the **DDE.POKE** subroutine. You must provide the name of the application, a topic recognised by that application, an item within that topic and the information to be sent. You must also provide a variable in which the completion status of the subroutine can be returned.

The following example adds a title to the beginning of a Microsoft Word document:

```
CALL DDE.POKE("winword"❶, ...
              "STAFF.DOC"❷, ...
              "\StartOfDoc"❸, ...
              "Staff Expenses":CHAR(13)❹, ...
              ERR❺)
```

❶ This is Microsoft Word's DDE application name.

❷ This parameter specifies the topic – in this case, the name of the Word document.

❸ "\StartOfDoc" is a built-in bookmark that marks the beginning of any Word document. Word recognises bookmark names as DDE items within document topics.

❹ This parameter specifies the information to be sent – in this case, the text "Staff Expenses" followed by a carriage return. Because the item specified is the \StartOfDoc bookmark, this text will be inserted at the beginning of the document.

❺ This parameter is a variable in which the completion status of the subroutine will be returned.

CAUTION

If the specified item is not recognised by the server application, the DDE.POKE operation will fail. No error code will be returned, however.

Sending Commands

The **DDE.EXECUTE** subroutine allows you to send one or more commands to the server application. You must provide the name of the application, a topic recognised by that application and a string containing one or more server commands. You must also provide a variable in which the completion status of the subroutine can be returned.

Many Microsoft applications, such as Excel and Word for Windows, recognise macro language statements and functions (refer to the documentation for your application for details of the commands you can use). For example, in Word for Windows, the WordBASIC command that creates a new document is FileNew. To send the same command via DDE, you would use the following:

```
CALL DDE.EXECUTE("winword"❶, ...
           "System"❷, ...
           '[FileNew .NewTemplate = 0, .Template = "Normal"]'❸, ...
           ERR❹)
```

❶ This is Microsoft Word's DDE application name.

❷ This parameter specifies the topic. This would normally be the name of the Word document, but in this case we are creating a new document, so we can use the System topic.

❸ This is the command to be executed. Note that Microsoft Word requires that each command received via DDE be enclosed in square brackets.

Note also that because WordBASIC requires the template name in the form of a string enclosed in double quotes, the command string has been constructed within single quotes.

❹ This parameter is a variable in which the completion status of the subroutine will be returned.

If required, you can send several commands at once. Each separate command must be enclosed in square brackets. For example, the following tells Microsoft Word to create a new document and then close it without saving:

```
COMMAND = '[FileNew .NewTemplate = 0, .Template = "Normal"]':...
          '[FileClose 2]'
CALL DDE.EXECUTE("winword", "System", COMMAND, ERR)
```

Note that there must be no spaces between bracketed commands in a single **DDE.EXECUTE** call, or an error will occur.

The preceding example could have been sent as two separate commands as follows:

```
COMMAND = '[FileNew .NewTemplate = 0, .Template = "Normal"]'
CALL DDE.EXECUTE("winword", "System", COMMAND, ERR)
COMMAND = '[FileClose 2]'
CALL DDE.EXECUTE("winword", "System", COMMAND, ERR)
```

Using Advise Links

An advise link is different to the DDE operations described above, in that you must establish it before you can use it to obtain data, and you must terminate it when it is no longer required.

Establishing a Link

You establish an advise link by calling the **DDE.OPENADVISE** subroutine. You must provide the name of the application, a topic recognised by that application and an item within that topic. You must also provide two variables in which a link identifier and the completion status of the subroutine can be returned. For example, the following establishes a link to a cell in an Excel spreadsheet:

```
CALL DDE.OPENADVISE("Excel"❶, ...
                    "C:\EXCEL\EXAMPLES\BUDGET.XLS"❷, ...
                    "R1C1"❸, ...
                    LINKIDENT❹, ...
                    ERR❺)
```

❶ This is Microsoft Excel's DDE application name.

❷ This parameter specifies the required topic – in this case the name of an Excel spreadsheet.

❸ "R1C1" identifies the first cell in the spreadsheet as the item for the DDE conversation.

❹ This parameter is a variable in which a link identifier will be returned. This identifier must be used when obtaining data from the link, and when closing it again.

❺ This parameter is a variable in which the completion status of the subroutine will be returned. A return value of zero indicates successful completion.

Obtaining Data via a Link

Once the link has been established, you can call **DDE.ADVISE** to obtain the item data. **DDE.ADVISE** requires the link identifier and two variables: one to return the data, and a second to return the status of the link.

Note: Because only *changed* data is returned, **DDE.ADVISE** will not normally return any data when the link is first established. To obtain the initial state of the item, use the **DDE.PEEK** subroutine.

The following obtains data from the link established in the previous example:

```
CALL DDE.ADVISE(LINKIDENT❶, DATA❷, STATUS❸)
```

❶ This is the link identifier returned by **DDE.OPENADVISE**.

❷ This parameter is a variable in which to return the link data.

❸ This parameter is a variable in which to return the status of the link. There are three return values that indicate success:

ADV.NODATA	The conversation item has not changed since **DDE.ADVISE** was last called.
ADV.MOREDATA	The conversation item has changed more than once since **DDE.ADVISE** was last called. The data returned is the result of the first change. To obtain the result of the next change, call **DDE.ADVISE** again.
ADV.LASTDATA	The conversation item has changed once since **DDE.ADVISE** was last called. The data returned is the result of this change.

Any other value indicates that an error has occurred. Note that if the returned status is **ADV.NODATA**, the data returned should be ignored.

Your application should call **DDE.ADVISE** periodically to obtain updated data. This could be done in a loop or, if your application has an event loop, by creating a timer and calling **DDE.ADVISE** each time a timer message is received.

Terminating a Link

When your no longer need your advise link, you should close it by calling the **DDE.CLOSEADVISE** subroutine. You must supply the identifier for the link you wish to close and a variable in which to return a status code. The following shows how to close the link used in the previous example:

```
CALL DDE.CLOSEADVISE(LINKIDENT❶, ERR❸)
```

❶ This is the link identifier returned by **DDE.OPENADVISE**.

❺ This parameter is a variable in which the completion status of the subroutine will be returned.

Example DDE Application

The UIMS-EXAMPLES file contains a simple example application – DDE – that demonstrates the use of all the DDE subroutines. This example does not require a header file or a resource script.

Chapter 14
Hybrid Applications

This chapter describes how to enhance existing applications by creating UIMS modules with their own message loops. It also shows you how to add a UIMS dialog box to the NewView application that was developed in Chapter 3.

Introduction

Chapter 3 describes how you can use NewView to add a graphical user interface to an existing character-based application. There are limits to what can be achieved with NewView, however, and you might like to use some of the UIMS features described in the previous chapters.

An application in which the user interface combines the character-based elements used on normal terminals with UIMS graphical objects is called a Hybrid application. A NewView application is the simplest form of hybrid application, in that it does not require a message loop. In writing more complex hybrid applications, you will need to provide one or more message loops for the UIMS elements of the user interface, while retaining the conventional PRINT, CRT and INPUT statements for the character-based elements.

When writing a hybrid application, you should not assume that the user will have UIMS, or even RealLink available. In the NewView example described in Chapter 3, the application carries out different actions depending on whether or not UIMS is available – some commands are only provided if it is. If you are writing UIMS versions of existing routines, character-based alternatives will already exist, but for new features you must decide whether to write both UIMS and character-based versions.

UIMS Modules

It is best to keep UIMS features separate from the character-based code. This can be done by creating UIMS modules – subroutines dedicated to the UIMS processing. These subroutines can be part of the main program or separately cataloged.

The components of a UIMS module are similar to those of a complete UIMS application:

- An initialisation routine to save the main application's environment and set up the UIMS environment required by the module.
- A routine to create the resources needed by the module. This can be done dynamically, as part of the initialisation procedure, or can simply consist of loading a pre-defined resource file.
- A message loop to process messages relating to the module.
- A closing routine to destroy the module's resources and restore the environment to that required by the calling application.

Note that in most cases it will not be necessary to sign on to UIMS, since this will have been done by the calling application. Instead, the main application should pass its application context handle to the UIMS module (either via a global or common variable, or by passing it as a parameter to a cataloged subroutine), together with any other parameters such as the handle of the main App window.

Resources

The resources required by a UIMS module should, in general, be kept separate from those in the main application. However, this can result in a delay before any window or dialog box is displayed, due to the time taken to create the resources. If this is unacceptable, you could load the resources at the same time as those for the main application, but keep them unmapped until needed. This will, however, increase the time taken to start the main application.

Note that the identifier values used by a UIMS module's resources must be unique. If the values used conflict with those in the main program, the resources will not load.

Combining Terminal Data and Messages

Character-based applications transfer data to and from the terminal as streams of characters. For example, when you type the command 'WHO' and press RETURN, the characters 'W', 'H', 'O', CHAR(13) are sent to the host. When a UIMS application is running, however, these key operations are converted in to keypress messages that are passed to the application by means of the **GetMsg** subroutine. A hybrid application combines these two modes of operation, with some routines using character-based data, while others have a message loop and therefore expect to receive keypress messages.

For a hybrid application to function correctly, RealLink must always be in the correct mode of operation, returning character-based data or UIMS messages as required. It is important that the correct operating mode is selected – if **GetMsg** is called while character mode is active, no key presses will be returned to the application.

In most cases this gives no problems – using character-based data switches RealLink into character mode, and calling a UIMS subroutine selects message mode. There are some UIMS subroutines that do not select message mode, however, and it is, of course, possible to call **GetMsg** without first calling another UIMS subroutine.

The **SetUimsMode** subroutine is provided to overcome this problem. If you are not receiving the keypress messages that you expect and think that RealLink may be in the wrong mode, you can call **SetUimsMode** before calling **GetMsg**.

To help you decide whether to use **SetUimsMode**, the following sections list the circumstances in which character mode and message mode are selected.

Character Mode

Character mode is selected when an application uses any of the following:

- Any of the NewView subroutines.
- The **Execute**, **SendKeys**, or **SystemCommand** subroutines.

- DATA/BASIC commands that send data to or receive data from the terminal (PRINT, CRT, etc.).

Message Mode

All UIMS subroutines select message mode, with the exception of the following:

- **GetMsg**
- **BitTest**
- **HiByte**
- **LoByte**

An Example Application: MENUDLG

The MENUDLG application illustrates the concepts described above. It is similar to the MENUNV NewView application, but the Options menu has an additional command, Send Message. This displays the same dialog box as the SendMsg application described in Chapter 10. The Send Message command is implemented as a self-contained cataloged subroutine with its own message loop.

To create the MENUDLG application, copy and rename the following source files:

Source	New name
MENUNV.H	MENUDLG.H
MENUNV.UCL	MENUDLG.UCL
MENUNV.DB	MENUDLG.DB
SENDMSG.H	SMSGDLG.H
SENDMSG.UCL	SMSGDLG.UCL
SENDMSG.DB	SMSGDLG.DB

Then make the following changes:

1. Make the changes described in the section on using the Generic application as a template in Chapter 4.
2. Add new constant definitions to the application's header file.
3. Add a Send Message item to the application's Options menu.
4. Add the Send Message command to the application's Option menu group
5. Add the Send Message command to the application's response loop.
6. Delete unwanted constant definitions from the SMSGDLG subroutine's header file, and change the remaining definitions so that they do not conflict with those used in the main application.
7. Delete the definition of the App window from the subroutine's resource script.
8. Change the definition of the Send Message dialog box so that it will be displayed when it is attached to its parent.

9. Add a SUBROUTINE statement at the beginning of the SMSGDLG subroutine.

10. Delete unnecessary code from the SMSGDLG subroutine source file.

11. Change the subroutine's main routine to set up and display the dialog box.

12. Change the initialisation procedure in the subroutine's main routine.

13. Change the subroutine's exit procedure to restore the state required by the main application.

14. Change the commands that unmap the dialog box, so that they cause the subroutine to return to the main application instead.

15. Compile the resource files and the DATA/BASIC programs.

Add New Constant Definitions

You will need identifiers for the additional resources defined in the application's resource script. These must be available to both the resource script and the DATA/BASIC source, so add the following to the header file, MENUDLG.H:

```
EQU APPMENUMSG     TO 153

EQU MSG.RESP     TO 'MSG'
```

Define Application Resources

To use the Send Message command, the user must be able to select it from a menu. Find the definition of the Option menu in the MENUDLG.UCL resource script and change it to the following:

```
* The Options menu has Diary, Calculator, Swap and
* Send Message commands
MENU = APPMENUOPTIONS
{
  TITLE = '&Options'
  ENABLED = FALSE
  CHILDREN = '&Diary'        = APPMENUDIARY,
             '&Calculator'   = APPMENUCALC,
             '&Swap strings' = APPMENUCHANGE,
             'Send &Message' = APPMENUMSG
}
```

Note that the line that defines the Swap strings item must now end with a comma.

Modify the Option Menu Group

In a NewView application, a menu command must be part of a NewView group. Find the MENUITEM.GROUPS subroutine in the file MENUDLG.DB, and change the definition of group 2 to the following:

```
* Responses for group 2 (Options menu)
* This contains Diary, Calculator, Swap and Send Message commands
MENU2GRP.RESP = DIARY.RESP:CRET:AM:...
                CALC.RESP:CRET:AM:...
                SWAP.RESP:CRET:AM:...
                MSG.RESP:CRET
* Create the group
CALL CreateNVContactGroup(CONTEXT, ...
                          MENU2GRPID, ...
                          APPMENUDIARY, ...
                          4, ...
                          MENU2GRP.RESP, ...
                          ERR)
```

Modify the Response Loop

The MENUDLG application must display a dialog box in response to the Send Message command. To do this, you will need an additional CASE statement in the application's response loop.

Find the CASE structure within the response loop in the main routine of MENUDLG.DB, and add the following:

```
CASE ANS = MSG.RESP
  IF UIMS.CAPABLE THEN
    CALL SMSGDLG(CONTEXT, APPWIN)
    CALL SetContactFocus(CONTEXT, CHILDWIN, ERR)
  END ELSE
    * otherwise the option is invalid
    ERRMSG = "Invalid entry : ":ANS ;* error message
    GOSUB ERRSUB                ;* handle the error
  END
  ANS = 0         ;* continue with this menu
*
```

Because MENUDLG will already be signed on to UIMS, you must pass the handle of the application context to the SMSGDLG subroutine. The dialog box will be attached to the main application window, so the handle of this must also be supplied. Note that a dialog box can only be made a child of an App window or the application context.

When the SMSGDLG subroutine returns, the focus must be returned to the terminal window, CHILDWIN. If this is not done, the user will be unable to enter selections from the keyboard.

Modify Constant Definitions

The file SMSGDLG.H will define the constants used in the SMSGDLG subroutine. Because each object used in a UIMS application must have a unique identifier, you will need to delete unused constants and change the values of those for the Send Message dialog box. If this is not done, it will not be possible to load the dialog resources.

Edit the subroutine header file, SMSGDLG.H, and delete the following lines:

```
EQUATE Win1         TO 10

* Help menu

EQUATE HelpAbout    TO 121

* Utilities menu

EQUATE UtilSendMsg TO 101
```

Then change the remaining constant definitions to the following:

```
EQUATE Dialog1      TO 1000
EQUATE Text1        TO 1010
EQUATE Edit1        TO 1011
EQUATE Text2        TO 1020
EQUATE List1        TO 1022
EQUATE ExGroup      TO 1030
EQUATE Option1      TO 1031
EQUATE Option2      TO 1032
EQUATE Option3      TO 1033
EQUATE Check1       TO 1040
EQUATE SendButton   TO 1050
EQUATE CancelButton TO 1051
EQUATE HelpButton   TO 1052
```

Note: The values listed above have been chosen because none of the identifiers used in the MENUDLG application are greater than 1000.

Modify the Subroutine's Resources

The SendMsg application, on which the SMSGDLG subroutine is based, has its own application window. This is not required in the subroutine and must be removed from the resource script. Edit SMSGDLG.UCL and delete the following lines from the beginning of the file:

```
APPWINDOW = Win1
{
  TITLE = 'SendMsg Example Application'
  STYLE = CLOSABLE, SIZABLE, MOVABLE, ICONISABLE
  BDRSTYLE = BORDER
  POSITION = 125, 167
  SIZE = 500, 417
  MENUBAR = 0
  {
    MENU = 0
    {
      TITLE = '&Utilities'
      CHILDREN = 'Send &Message...' = UtilSendMsg
    }

    MENU = 0
    {
      TITLE = '&Help'
      CHILDREN = '&About SendMsg...' = HelpAbout
    }
  }
```

Then move to the end of the file and delete the final closing brace.

In the SendMsg application, the Send Message dialog box is created as a child of the main App window, but is kept in an unmapped state until required. In the MENUDLG application, the dialog resources are only loaded when needed; the dialog can therefore be created in the mapped state. To do this, delete the following line from the definition of the Dialog Box:

```
MAPPED = FALSE
```

Add the SUBROUTINE Statement

You will need to convert SMSGDLG into a catalogued subroutine. Add the following line at the very beginning of the source file:

```
SUBROUTINE SMSGDLG(CONTEXT, WINDOW)
```

The CONTEXT and WINDOW parameters will allow the main application to pass the handles of the application context and the main App window to the subroutine.

You must also make sure that the subroutine can return to the main application. Find the following line in the source file:

```
STOP                                    ;* return to TCL
```

and replace it with:

```
* Return to the main application
RETURN
```

Delete Unnecessary Code

Much of the SendMsg application is concerned with processing messages for the main App window and its menus. This code will not be needed in the SMSGDLG subroutine and can be deleted.

You can delete the following routines from the file SMSGDLG.DB:

- HANDLE.WIN1.MESSAGES
- HANDLE.WIN1.MENU
- SHOW.ABOUT.BOX

You can also delete the Win1 case in the message loop. Find and delete the following lines:

```
CASE MSG.WINDOW = Win1
  GOSUB HANDLE.WIN1.MESSAGES
```

The main application is using the terminal window, so your subroutine will not need to hide it. Find the following lines and delete them:

```
* Hide the RealLink window
* This can be done at any time, but it's more reassuring to the
* user if we wait until the application's window has appeared.

CALL SetTeWindow(0, 0, UIMS.NONE, ERR)
```

Set up and Display the Dialog Box

You should already have changed the call to **LoadAppRes** to load the subroutine's resources. Once the resources have been loaded, the dialog box must be set up and then displayed by attaching it as a child of the main application's App window. Find the following lines:

```
* Add Win1 as a child of the context returned by the SignOn call.
* This has the effect of 'drawing' Win1 and its children.
```

```
CALL AddChild(CONTEXT, CONTEXT, -1, Win1, ERR)
```

and replace them with:

```
* Set up the dialog
GOSUB SETUP.DIALOG1

* Add Dialog1 as a child of the context returned by the SignOn call.
* This has the effect of 'drawing' Dialog1 and its children.

CALL AddChild(CONTEXT, WINDOW, -1, Dialog1, ERR)
```

The handle of the application's App window will be passed to the subroutine in the WINDOW variable.

Initialise the Subroutine

The initialisation procedure for a UIMS subroutine is different to that for a complete UIMS application. Where the main application uses NewView, it will already be signed on to UIMS and the coordinate mode will have been set. However, you may need to change the error handling mode, and the primary and secondary event masks.

Find the following lines:

```
* Sign on to UIMS

CALL InitialiseUims
CALL SignOn("SENDMSG", CONTEXT)
IF NOT(CONTEXT) THEN
  PRINT "Failed to Signon"
  STOP
END

* Screen positions and contact sizes will be specified in pixels

CALL SetCoordMode(CONTEXT, UIMS.COORD.GRAPHIC, ERR)
```

and replace them with:

```
CALL SetSync(CONTEXT, FALSE, ERR)  ❶

* Enable messages
CALL GetEventMask(CONTEXT, CONTEXT, EVENTMASK)
OLD.EVENTMASK = EVENTMASK  ❷
```

```
CALL BitTest(EVENTMASK, UIMS.EM.NOTIFY, ENABLED) ❸
IF NOT(ENABLED) THEN EVENTMASK = EVENTMASK + UIMS.EM.NOTIFY
CALL BitTest(EVENTMASK, UIMS.EM.MOTION, ENABLED)
IF ENABLED THEN EVENTMASK = EVENTMASK - UIMS.EM.MOTION ❹
CALL SetEventMask(CONTEXT, CONTEXT, EVENTMASK, ERR)

CALL SetSecondaryEventMask(CONTEXT, EVENTMASK, FALSE, FALSE, ERR) ❺
```

❶ Because the calling, NewView application does not have a message loop, errors are handled synchronously. In the SMSGDLG subroutine, it will be simpler to handle them asynchronously.

❷ This line saves the main application's event mask for restoration when the subroutine returns.

❸ The NewView event mask used by the main application disables Notify messages. If errors are to be handled asynchronously, Notify messages must be enabled.

❹ The NewView event mask used by the main application enables mouse Motion messages. These are not required by the SMSGDLG subroutine.

❺ Since the main application does not have a message loop, the secondary event mask is set to stop any messages reaching the application. The SMSGDLG subroutine, however, must receive all messages that are enabled by the primary mask. The secondary event mask is therefore set to the same value as the primary mask.

Restore the Previous State

On returning to the main application, the subroutine must restore the previous error handling, and primary and secondary event mask states. The dialog box must also be destroyed before returning.

Find the following lines in the main routine:

```
* re-display the RealLink window
CALL SetTeWindow(0, 0, TE.SHOWWIN, ERR)
CALL SignOff(CONTEXT, ERR)              ;* sign off from UIMS
```

and replace them with:

```
* Restore synchronous error handling
CALL SetSync(CONTEXT, TRUE, ERR)
* Restore the old event mask
CALL SetEventMask(CONTEXT, CONTEXT, OLD.EVENTMASK, ERR)
```

```
* Prevent any more messages reaching the application
CALL SetSecondaryEventMask(CONTEXT, 0, FALSE, FALSE, ERR)
* Destroy the dialog box
CALL Destroy(CONTEXT, Dialog1, ERR)
```

Change the Unmap Commands

In the SendMsg application, when the Send Message dialog box is no longer needed, it is hidden by setting it to the unmapped state. The SMSGDLG subroutine must instead return to the main application.

The Send and Cancel button commands, and the Close command on the System menu, must each be changed. Find every occurrence of the following lines:

```
* Hide the dialog box
CALL UnMap(CONTEXT, Dialog1, ERR)
```

and replace them with:

```
* close the application
USER.WANTS.TO.EXIT = TRUE
```

Compile

When you have made these changes, you can compile the resource scripts and DATA/BASIC programs as described in Chapter 4 for the Generic application. When you run the MENUDLG application, the Options menu will initially be disabled. Select any item from the main menu and then select the Send Message command from the Options menu on the menu bar. You will see a dialog box similar to that described in Chapter 10 for the SendMsg application. Type a message in the edit box, select a destination and then click Send to send the message.

Appendix A
NewView Examples

This appendix gives the source code listings for the MENUEX and MENUNV programs described in Chapter 3.

MENUEX

DATA/BASIC Source Code

```
***************************************************************
*
* PROGRAM: MENUEX
*
* PURPOSE: Example application for demonstating NewView conversion
*
* ROUTINES:
*   Main routine - initialises the application and processes user input
*   BUILD - builds a menu to display
*   ERRSUB - processes errors
*
***************************************************************


        *** Constant definitions ***

* responses common to all menus
EQU MAIN.RESP TO 'M'      ;* return to main menu
EQU BACK.RESP TO 'E'      ;* return to previous menu
EQU EXIT.RESP TO 'OFF'    ;* return to TCL

* item attributes in the menu definition file
EQU MOPT.NO TO 1          ;* item number or identifier
EQU MCOL TO 2             ;* screen column
EQU MROW TO 3             ;* screen row
EQU MTITLE TO 4           ;* item title
EQU MLINK TO 5            ;* links to sub-menus
EQU MHEADING TO 6         ;* menu heading


* Open the menu definitions file
OPEN "MENUDEFS" TO MENUDEFS ELSE CRT "No MENUDEFS file"; STOP

ID = "MAIN"               ;* start with the main menu
PREVID = ''               ;* initialise history
PROMPT " "                ;* we don't want a prompt character for input statements
```

```
        *** Main loop ***
OK = 1                     ;* initialise loop control variable
LOOP WHILE OK DO
   GOSUB BUILD                 ;* build the selected menu
   IF SCR # '' THEN
      CRT SCR:                 ;* display the menu
      INS ID BEFORE PREVID<1> ;* add this menu to the history
   END

           *** Response loop ***
   ANS = 0                     ;* initialise loop control variable
   LOOP WHILE ANS = 0 DO

      * get the user's selection
      CRT @(SELCOL,SELROW):    ;* position the cursor
      CRT @(-4)                ;* clear previous selection
      CRT @(SELCOL,SELROW):    ;* reposition the cursor
      INPUT ANS,10:            ;* get the user's choice

      BEGIN CASE
*
      CASE ANS = BACK.RESP     ;* go back to the previous menu
         DEL PREVID<1>            ;* delete current id from history
         ID = PREVID<1>           ;* get the id of the previous menu
         DEL PREVID<1>            ;* delete this id as well
         IF ID = '' THEN ID = "MAIN" ;* if already at main menu, stay there
*
      CASE ANS = MAIN.RESP     ;* back to the main menu
         ID = "MAIN"              ;* set the current id
         PREVID = ''              ;* re-initialise the history
*
      CASE ANS = EXIT.RESP     ;* return to TCL
         OK = 0                   ;* clear the control variable for the main loop
*
      CASE ANS >= START.OPT.NO AND ANS <= END.OPT.NO ;* valid menu item
         ID = MENUREC<MLINK,ANS> ;* point to the selected sub-menu
*
      CASE 1                   ;* the response was none of the above
         ERRMSG = "Invalid entry :":ANS ;* error message
         GOSUB ERRSUB                    ;* handle the error
         ANS = 0           ;* clear the loop control so that we continue with this menu
*
      END CASE
```

```
*
   REPEAT               ;* end of response loop

REPEAT               ;* end of main loop

        *** Exit program ***

CRT @(-1)            ;* clear the screen
STOP                 ;* return to TCL


***************************************************************
*
* SUBROUTINE: BUILD
*
* PURPOSE: Build a menu for display
*
* COMMENTS:
*    ID specifies the menu required. The menu structure is read
*    from the menu definition file.
*    On exit the following variables are set:
*      SCR            - The completed menu for display
*      START.OPT.NO   - The number of the first item
*      END.OPT.NO     - The number of the last item
*      SELCOL, SELROW - The screen position for the INPUT
*                         statement in the main loop
*
***************************************************************

BUILD:
   SCR = ''                ;* clear any previous menu

   * read the specified menu from the open menu definition file
   READ MENUREC FROM MENUDEFS,ID ELSE ERRMSG = "No menu record" ; GOSUB ERRSUB ; RETURN

           *** Start building the menu ***
   SCR = @(-1)             ;* clear the screen
   * position and text for menu heading
   SCR = SCR:@(MENUREC<MHEADING,1>,MENUREC<MHEADING,2>):MENUREC<MHEADING,3>

   START.OPT.NO = 0; END.OPT.NO = 0 ;* initialise the range of valid items
   OPT.NO = 0                       ;* current item
   X = 0                           ;* start counting items from 0
   * for each item on the menu
   LOOP X = X+1 WHILE MENUREC<MOPT.NO,X> # '' DO

      * fetch item parameters
      COL = MENUREC<MCOL,X>          ;* screen column
      ROW = MENUREC<MROW,X>          ;* screen row
      TITLE = MENUREC<MTITLE,X>      ;* item title
      OPT.NO = MENUREC<MOPT.NO,X>    ;* item number or identifier

      * if the item has a number
      IF OPT.NO MATCHES '0N' THEN
         * if not already set, set the start of the valid range
         IF START.OPT.NO = 0 THEN START.OPT.NO = OPT.NO

         END.OPT.NO = OPT.NO ;* extend the end of range to include this item
      END

      IF OPT.NO = 'S' THEN
         * if the item is the selection prompt
         *   set the position for user input
         SELCOL = COL+LEN(TITLE); SELROW = ROW

         *   append the position and text for the selection prompt to the menu
         SCR = SCR:@(COL,ROW):TITLE

      END ELSE
         * otherwise append the position and text for this item to the menu
         SCR = SCR:@(COL,ROW):OPT.NO:@(COL+5,ROW):TITLE
      END
   REPEAT ;* repeat for the next item

RETURN


***************************************************************
*
* SUBROUTINE: ERRSUB
*
* PURPOSE: Display an error message and wait for a response
*
* COMMENTS:
*    The ERRMSG variable contains the message to display
```

```
*
***************************************************************

ERRSUB:
   CRT @(LEN(ERRMSG)+4,23):                    ;* position the cursor
   CRT @(-128):                                ;* video attributes off
   * print the error message in reverse video
   CRT @(0,23):@(-132):" ":ERRMSG:" ":@(-128):
   INPUT ANS,1:                                ;* wait for the user to respond
   CRT @(0,23):@(-4):                          ;* clear the error message
RETURN


***************************************************************
END
```

Menu Definition File

The following items must be created on the host in a file called MENUDEFS. This must be in the same account as the compiled and cataloged version of MENUEX.

In the item details that follow, REALITY value marks (X'FD') are shown as closing square brackets ( ]).

MAIN Item

Attribute 1:	`1]2]3]4]5]6]7]8]S`
Attribute 2:	`25]25]25]25]25]25]25]25]25`
Attribute 3:	`5]7]9]11]13]15]17]19]22`
Attribute 4:	`Personal Updates]Personal Enquiries]Background Files]Reporting Facilities]Bulk Amendments] Daily/Weekly/Monthly Procedures]Additional Facilities]System Administration]Selection or 'OFF':`
Attribute 5:	`SUB1]SUB2]SUB3]SUB4`
Attribute 6:	`30]2]MAIN MENU`

SUB1 Item

Attribute 1:	`1]2]3]4]5]6]7]8]9]10]S`
Attribute 2:	`1]1]1]1]1]40]40]40]40]40]25`
Attribute 3:	`5]7]9]11]13]5]7]9]11]13]22`

	Attribute 4:	`Personal (i)]Personal (ii)]Pay Details]Benefits]Private Health Scheme]Additional Details]Qualifications] Training Requirements]Training History]Career History]Selection, 'M' or 'E':`
	Attribute 5:	`SUB5]`
	Attribute 6:	`30]1]PERSONNEL UPDATES`
SUB2 Item	Attribute 1:	`1]2]3]4]5]6]7]8]9]10]S`
	Attribute 2:	`1]1]1]1]1]40]40]40]40]40]25`
	Attribute 3:	`5]7]9]11]13]5]7]9]11]13]22`
	Attribute 4:	`Personal (i)]Personal (ii)]Pay Details]Benefits]Private Health Scheme]Additional Details]Qualifications] Training Requirements]Training History]Career History] Selection, 'M' or 'E':`
	Attribute 5:	
	Attribute 6:	`30]1]PERSONNEL ENQUIRIES`
SUB3 Item	Attribute 1:	`1]2]3]4]5]6]S`
	Attribute 2:	`25]25]25]25]25]25]25`
	Attribute 3:	`8]10]12]14]16]18]22`
	Attribute 4:	`Standard Narrative Tables]Bespoke Narrative Tables] User-Defined Tables]Job Details]Establishment Structure]Reasons for Change]Selection, 'M' or 'E':`
	Attribute 5:	
	Attribute 5:	`30]1]BACKGROUND FILES`
SUB4 Item	Attribute 1:	`1]2]3]4]5]S`
	Attribute 2:	`25]25]25]25]25]25`
	Attribute 3:	`8]10]12]14]16]22`

	Attribute 4:	`Report Generator]Standard Letters]Record Sheets]English Reports]Bonus Procedures]Selection, 'M' or 'E':`
	Attribute 5:	`MAIN]MAIN]MAIN]MAIN]`
	Attribute 6:	`30]1]REPORTING FACILITIES`
SUB5 Item	Attribute 1:	`1]2]3]4]5]6]7]8]S`
	Attribute 2:	`1]1]1]1]40]40]40]40]25`
	Attribute 3:	`5]7]9]11]5]7]9]11]22`
	Attribute 4:	`Personal]Family]Financial]Medical History]Contacts] Relatives]Travel]Organizations]Selection, 'M' or 'E':`
	Attribute 5:	
	Attribute 6:	`30]1]PERSONAL DETAILS`

MENUNV

DATA/BASIC Source Code

The DATA/BASIC source can be created on the host with a REALITY text editor (ED or SE). Alternatively it can be created on the PC with a text editor such as Windows Notepad, and then copied onto the host with one of the RealLink file transfer utilities (LanFTU or HOST-WS).

```
***************************************************************
*
* PROGRAM: MENUNV
*
* PURPOSE: NewView example application
*
* ROUTINES:
*   Main routine    - initialises the application and processes
*                     user input
*   BUILD           - builds a menu to display
*   ERRSUB          - processes errors
*   SETUP.NEWVIEW   - signs on to UIMS and sets up the NewView
*                     environment
*   BUTTON.GROUPS   - creates button groups and assigns response
*                     strings to them
*   MENUITEM.GROUPS - creates menuitem groups and assigns
*                     response strings to them
*   SIGNOFF         - tidies up UIMS and NewView before returning
*                     to TCL
*   CREATE.HOTSPOTS - creates the hotspots for a menu
*   SWAP.BUTTONS    - demonstrates how NewView buttons can be changed
*                     in response to a menu selection
*   CHANGE.MENUBAR  - demonstrates how the commands on the application's
*                     menu bar can be changed, according to the menu
*                     displayed.
*
***************************************************************


*** NewView additions ***

* UIMS and RealLink constant definitions
INCLUDE UIMSDEFS FROM UIMS-TOOLS
INCLUDE UIMSCOMMON FROM UIMS-TOOLS
INCLUDE RFWDEFS FROM UIMS-TOOLS
```

```
* application-specific constant definitions
INCLUDE MENUNV.H

*** End of NewView additions ***


        *** Constant definitions ***

* responses common to all menus
EQU MAIN.RESP TO 'M'      ;* return to main menu
EQU BACK.RESP TO 'E'      ;* return to previous menu
EQU EXIT.RESP TO 'OFF'    ;* return to TCL

* item attributes in the menu definition file
EQU MOPT.NO TO 1          ;* item number or identifier
EQU MCOL TO 2             ;* screen column
EQU MROW TO 3             ;* screen row
EQU MTITLE TO 4           ;* item title
EQU MLINK TO 5            ;* links to sub-menus
EQU MHEADING TO 6         ;* menu heading


* Open the menu definitions file
OPEN "MENUDEFS" TO MENUDEFS ELSE CRT "No MENUDEFS file"; STOP

*** NewView additions ***

* Try to initialise UIMS. This will tell us whether the terminal and
* account support UIMS.
UIMS.CAPABLE = FALSE   ;* assume no UIMS support until we've tried
* Now see if the InitialiseUims subroutine exists. If it does, call it.
OPEN "MD" TO ACCMD THEN
   READ ACCREC FROM ACCMD,"InitialiseUims" THEN CALL InitialiseUims
   CLOSE ACCMD
END

* If UIMS is supported, we can set up UIMS and NewView
IF UIMS.CAPABLE THEN GOSUB SETUP.NEWVIEW

*** End of NewView additions ***

ID = "MAIN"               ;* start with the main menu
PREVID = ''               ;* initialise history
```

```
PROMPT " "                ;* we don't want a prompt character for input statements

*** NewView additions ***

HOTSPOTS = 0              ;* non-zero once NewView hotspots have been created
SWITCH = 0                ;* buttons haven't been swapped (Change command)

*** End of NewView additions ***


        *** Main loop ***
OK = 1                    ;* initialise loop control variable
LOOP WHILE OK DO
   GOSUB BUILD                ;* build the selected menu
   IF SCR # '' THEN
      CRT SCR:   ;* display the menu

      *** NewView additions ***

      IF UIMS.CAPABLE THEN GOSUB CREATE.HOTSPOTS   ;* create its hotspots

      *** End of NewView additions ***

      INS ID BEFORE PREVID<1> ;* add this menu to the history
   END

           *** Response loop ***
   ANS = 0                    ;* initialise loop control variable
   LOOP WHILE ANS = 0 DO

      * get the user's selection
      CRT @(SELCOL,SELROW):   ;* position the cursor
      CRT @(-4)               ;* clear previous selection
      CRT @(SELCOL,SELROW):   ;* reposition the cursor
      INPUT ANS,10:           ;* get the user's choice

      BEGIN CASE
*
      CASE ANS = BACK.RESP    ;* go back to the previous menu
         DEL PREVID<1>           ;* delete current id from history
         ID = PREVID<1>          ;* get the id of the previous menu
         DEL PREVID<1>           ;* delete this id as well
         IF ID = '' THEN ID = "MAIN" ;* if already at main menu, stay there
```

```
*
      CASE ANS = MAIN.RESP    ;* back to the main menu
         ID = "MAIN"             ;* set the current id
         PREVID = ''             ;* re-initialise the history
*
      *** NewView additions ***
*
      CASE ANS = SWAP.RESP   ;* change the buttons around
         IF UIMS.CAPABLE THEN
            GOSUB SWAP.BUTTONS ;* if we have a UIMS terminal swap the buttons
         END ELSE
            * otherwise the option is invalid
            ERRMSG = "Invalid entry : ":ANS ;* error message
            GOSUB ERRSUB                ;* handle the error
         END
         ANS = 0         ;* clear the loop control so that we continue with this menu
*
      CASE ANS = CALC.RESP   ;* start the windows calculator
         IF UIMS.CAPABLE THEN
            GOSUB RUN.CALC
         END ELSE
            * otherwise the option is invalid
            ERRMSG = "Invalid entry : ":ANS ;* error message
            GOSUB ERRSUB                ;* handle the error
         END
         ANS = 0         ;* clear the loop control so that we continue with this menu
*
      *** End of NewView additions ***
*
      CASE ANS = EXIT.RESP    ;* return to TCL
         OK = 0                  ;* clear the control variable for the main loop
*
      CASE ANS >= START.OPT.NO AND ANS <= END.OPT.NO ;* valid menu item
         ID = MENUREC<MLINK,ANS> ;* point to the selected sub-menu
*
      CASE 1                  ;* the response was none of the above
         ERRMSG = "Invalid entry : ":ANS ;* error message
         GOSUB ERRSUB                  ;* handle the error
         ANS = 0           ;* clear the loop control so that we continue with this menu
*
      END CASE
*
   REPEAT               ;* end of response loop
   *** NewView additions ***

   * different menus have different commands available
   IF UIMS.CAPABLE THEN GOSUB CHANGE.MENUBAR

   *** End of NewView additions ***

REPEAT               ;* end of main loop

        *** Exit program ***

CRT @(-1)            ;* clear the screen

*** NewView changes ***

IF UIMS.CAPABLE THEN GOSUB SIGNOFF ;* tidy up UIMS and NewView

*** End of NewView changes ***

STOP                 ;* return to TCL


***************************************************************
*
* SUBROUTINE: BUILD
*
* PURPOSE: Build a menu for display
*
* COMMENTS:
*    ID specifies the menu required. The menu structure is read
*      from the menu definition file.
*    The HOTSPOTS variable specifies whether there is a hotspot
*      group already in existence. If there is, it must be
*      destroyed before the new one can be created.
*    On exit the following variables are set:
*      SCR            - The completed menu for display
*      START.OPT.NO   - The number of the first item
*      END.OPT.NO     - The number of the last item
*      SELCOL, SELROW - The screen position for the INPUT
*                         statement in the main loop
*      HOTSPOTS       - The number of hotspots to be created
*      XPOS, YPOS     - Hotspot positions
*      WIDTH, HEIGHT  - Hotspot sizes
*      RESP           - Hotspot response strings
*
***************************************************************

BUILD:
   SCR = ''                ;* clear any previous menu

   * read the specified menu from the open menu definition file
   READ MENUREC FROM MENUDEFS,ID ELSE ERRMSG = "No menu record" ; GOSUB ERRSUB ; RETURN

   *** NewView additions ***

   IF UIMS.CAPABLE THEN
      * destroy any previous hotspot group
      IF HOTSPOTS # 0 THEN CALL DestroyNVGroup(CONTEXT, HOTGRPID, ERR)

      * reset hotspot attribute arrays
      XPOS = '' ; YPOS = ''      ;* positions
      WIDTH = '' ; HEIGHT = ''   ;* sizes
      RESP = ''                  ;* responses

      HOTSPOTS = 0               ;* no hotspots now exist
   END

   *** End of NewView additions ***

           *** Start building the menu ***
   SCR = @(-1)             ;* clear the screen
   * position and text for menu heading
   SCR = SCR:@(MENUREC<MHEADING,1>,MENUREC<MHEADING,2>):MENUREC<MHEADING,3>

   START.OPT.NO = 0; END.OPT.NO = 0 ;* initialise the range of valid items
   OPT.NO = 0                       ;* current item

   X = 0                            ;* start counting items from 0
   * for each item on the menu
   LOOP X = X+1 WHILE MENUREC<MOPT.NO,X> # '' DO

      * fetch item parameters
      COL = MENUREC<MCOL,X>         ;* screen column
      ROW = MENUREC<MROW,X>         ;* screen row
      TITLE = MENUREC<MTITLE,X>     ;* item title
      OPT.NO = MENUREC<MOPT.NO,X>   ;* item number or identifier

      * if the item has a number
      IF OPT.NO MATCHES '0N' THEN
         * if not already set, set the start of the valid range
         IF START.OPT.NO = 0 THEN START.OPT.NO = OPT.NO

         END.OPT.NO = OPT.NO ;* extend the end of range to include this item
      END

      IF OPT.NO = 'S' THEN
         * if the item is the selection prompt
         *   set the position for user input
         SELCOL = COL+LEN(TITLE); SELROW = ROW

         *   append the position and text for the selection prompt to the menu
         SCR = SCR:@(COL,ROW):TITLE

      END ELSE
         * otherwise append the position and text for this item to the menu
         SCR = SCR:@(COL,ROW):OPT.NO:@(COL+5,ROW):TITLE

         *** NewView additions ***

         * the hotspot will be created when the menu is displayed,
         * but its attributes are set up now
         IF UIMS.CAPABLE THEN
            * each attribute is added to the appropriate dynamic array
            XPOS<-1> = COL ; YPOS<-1> = ROW             ;* position
            WIDTH<-1> = LEN(TITLE)+5 ; HEIGHT<-1> = 1   ;* size
            RESP<-1> = OPT.NO:CRET                      ;* response

            HOTSPOTS = HOTSPOTS+1   ;* increment the hotspot counter
         END

         *** End of NewView additions ***

      END
   REPEAT ;* repeat for the next item

RETURN
```

```
**************************************************************
*
* SUBROUTINE: ERRSUB
*
* PURPOSE: Display an error message and wait for a response
*
* COMMENTS:
*    The ERRMSG variable contains the message to display
*
**************************************************************

ERRSUB:
   IF UIMS.CAPABLE THEN
      CALL CreateMessageBox(CONTEXT, UIMS.INFO, "Error", ERRMSG, "", REPLY, ERR)
      * Set the focus back to the terminal window
      CALL SetContactFocus(CONTEXT, CHILDWIN, ERR)
   END ELSE
      CRT @(LEN(ERRMSG)+4,23):                  ;* position the cursor
      CRT @(-128):                              ;* video attributes off
      * print the error message in reverse video
      CRT @(0,23):@(-132):" ":ERRMSG:" ":@(-128):
      INPUT ANS,1:                              ;* wait for the user to respond
      CRT @(0,23):@(-4):                        ;* clear the error message
   END
RETURN


**************************************************************
*
* SUBROUTINE: SETUP.NEWVIEW
*
* PURPOSE: Signs on to UIMS and sets up the NewView environment
*
**************************************************************

SETUP.NEWVIEW:
   CALL SignOn("MENUNV", CONTEXT) ;* sign on to UIMS
   IF NOT(CONTEXT) THEN
      UIMS.CAPABLE = FALSE   ;* can't sign on, so no UIMS support
      ERRMSG = "Error - failed to signon to UIMS"
      GOSUB ERRSUB
      RETURN
   END
```

```
* set an event mask that will allow NewView the right events to process
CALL SetEventMask(CONTEXT, CONTEXT, UIMS.EM.NEWVIEW, ERR)
* disable unwanted events
CALL SetSecondaryEventMask(CONTEXT, 0, FALSE, FALSE, ERR)

* select synchronous error handling
CALL SetSync(CONTEXT, TRUE, ERR)

* RealLink runs in Graphics mode - so make the application do the same
CALL SetCoordMode(CONTEXT, UIMS.COORD.GRAPHIC, ERR)

* load resource file to create window with buttons etc.
CALL LoadAppRes(CONTEXT, "menunv.res", ERR)
IF ERR THEN
   * can't use UIMS without resources, but the application can continue without
   UIMS.CAPABLE = FALSE   ;* no UIMS support
   CALL SignOff(CONTEXT, ERR) ;* sign off from UIMS
   ERRMSG = "Cannot find MENUNV.RES resource file"
   GOSUB ERRSUB               ;* display the error message
   RETURN
END

* parent the AppWindow to the Context to make it visible
CALL AddChild(CONTEXT, CONTEXT, -1, APPWIN, ERR)

* make the child window the TE window
CALL SetTeWindow(CONTEXT, CHILDWIN, UIMS.NONE, ERR)
* and give it the focus
CALL SetContactFocus(CONTEXT, CHILDWIN, ERR)

* redirect system messages to a message box
CALL ReMapNVLine25(CONTEXT, TRUE, ERR)
IF ERR THEN ERRMSG = "Unable to redirect system messages" ; GOSUB ERRSUB

* initialise a hotspot attributes
XPOS = '' ; YPOS = ''       ;* positions
WIDTH = '' ; HEIGHT = ''    ;* sizes
RESP = ''                   ;* responses
HOTSPOTS = 0                ;* number of hotspots

* form a NewView button group and assign response strings to individual buttons
GOSUB BUTTON.GROUPS
```

```
   * form a NewView group for the menu items and assign response strings
   GOSUB MENUITEM.GROUPS
RETURN


***************************************************************
*
* SUBROUTINE: BUTTON.GROUPS
*
* PURPOSE: Create button groups and assign response strings
*
* COMMENTS:
*    There are two button groups: group 1, consisting of the Back
*    and Main buttons (buttons 1 and 2); and group 2, containing
*    all the rest. Group 2 is initially disabled.
*
***************************************************************

BUTTON.GROUPS:
   * responses for button group 1
   * these are attributes in a dynamic array
   * each response is terminated with a carriage return
   BUT1GRP.RESP = BACK.RESP:CRET:AM:MAIN.RESP:CRET
   * create the group
   CALL CreateNVContactGroup(CONTEXT, BUT1GRPID, BUT1, 2, BUT1GRP.RESP, ERR)

   * responses for button group 2: only buttons 9 & 10 are used
   BUT2GRP.RESP = AM:AM:AM:AM:AM:AM:OK.RESP:CRET:AM:SWAP.RESP:CRET
   * create the group
   CALL CreateNVContactGroup(CONTEXT, BUT2GRPID, BUT3, 8, BUT2GRP.RESP, ERR)
   * disable it
   CALL SetEnabledNVGroup(CONTEXT, BUT2GRPID, FALSE, ERR)

   * alternative responses for button group 1 (used when the buttons are swapped)
   BUT1GRP.RESP2 = MAIN.RESP:CRET:AM:BACK.RESP:CRET

   * map the button groups to make them visible
   CALL SetMappedNVGroup(CONTEXT, BUT1GRPID, TRUE, ERR)
   CALL SetMappedNVGroup(CONTEXT, BUT2GRPID, TRUE, ERR)
RETURN
```

```
**************************************************************
*
* SUBROUTINE: MENUITEM.GROUPS
*
* PURPOSE: Create menuitem groups and assign response strings
*
* COMMENTS:
*    There are three menus: File (containing Print commands and
*    Exit), Edit, and Options. The Print commands and the Edit
*    menu are managed by RealLink.
*    Note that the Options menu is disabled in the resource file
*
**************************************************************

MENUITEM.GROUPS:
   * Responses for group 1 (File menu)
   * Because the Print commands are managed by RealLink, this
   * group consists of only a single item: Exit
   MENU1GRP.RESP = EXIT.RESP:CRET
   * Create the group
   CALL CreateNVContactGroup(CONTEXT, MENU1GRPID, APPMENUEXIT, 1, MENU1GRP.RESP, ERR)

   * Responses for group 2 (Options menu)
   * This contains Diary, Calculator and Swap commands
   MENU2GRP.RESP = DIARY.RESP:CRET:AM:CALC.RESP:CRET:AM:SWAP.RESP:CRET
   * Create the group
   CALL CreateNVContactGroup(CONTEXT, MENU2GRPID, APPMENUDIARY, 3, MENU2GRP.RESP, ERR)
RETURN


**************************************************************
*
* SUBROUTINE: SIGNOFF
*
* PURPOSE: To tidy up UIMS and NewView before returning to TCL
*
* COMMENTS:
*    NewView contact and hotspot groups, if any, are destroyed. The
*    HOTSPOT variable specifies whether there are any hotspot groups
*    to be destroyed.
*
**************************************************************
```

```
SIGNOFF:
   * Destroy the hotspot group, if any.
   IF HOTSPOTS # 0 THEN CALL DestroyNVGroup(CONTEXT, HOTGRPID, ERR)

   * Destroy the two button groups
   CALL DestroyNVGroup(CONTEXT, BUT1GRPID, ERR)
   CALL DestroyNVGroup(CONTEXT, BUT2GRPID, ERR)
   * and the two menu-item groups
   CALL DestroyNVGroup(CONTEXT, MENU1GRPID, ERR)
   CALL DestroyNVGroup(CONTEXT, MENU2GRPID, ERR)

   * return TE functionality to the RealLink window
   CALL SetTeWindow(0, 0, TE.SHOWWIN, ERR)

   CALL SignOff(CONTEXT, ERR)   ;* sign off from UIMS
RETURN


***************************************************************
*
* SUBROUTINE: CREATE.HOTSPOTS
*
* PURPOSE: Create a hotspot for each menu item
*
* COMMENTS:
*    The HOTSPOTS variable contains the number of hotspots to create
*    The following dynamic arrays contain hotspot attributes:
*       XPOS, YPOS    - position
*       WIDTH, HEIGHT - size
*       RESP          - response string
*
***************************************************************

CREATE.HOTSPOTS:
   * create the menu's hotspots
   CALL CreateNVHotspotGroup(CONTEXT, HOTGRPID, HOTSPOTS, XPOS, YPOS, WIDTH, HEIGHT, ...
                            RESP, ERR)
   IF ERR # 0 THEN
      ERRMSG = "Failed to create Hotspot group : ":ERR ; GOSUB ERRSUB
   END
RETURN
```

```
***************************************************************
*
* SUBROUTINE: SWAP.BUTTONS
*
* PURPOSE: To demonstrate how NewView buttons might be changed in
*          response to menu input.
*
* COMMENTS:
*    Both the button titles and their responses must be changed.
*    The SWITCH variable indicates whether the buttons have
*    already been swapped.
*
***************************************************************

SWAP.BUTTONS:
   IF SWITCH = 0 THEN
      * change the titles of the buttons in group 1
      CALL TitledButtonSetTitle(CONTEXT, BUT1, "Main", ERR)
      CALL TitledButtonSetTitle(CONTEXT, BUT2, "Back", ERR)

      * change the button responses
      CALL ChangeNVContacts(CONTEXT, BUT1GRPID, BUT1, 2, BUT1GRP.RESP2, ERR)

      * set a flag to show that the buttons have been swapped
      SWITCH = 1

   END ELSE
      * change the titles of the buttons in group 1
      CALL TitledButtonSetTitle(CONTEXT, BUT2, "Main", ERR)
      CALL TitledButtonSetTitle(CONTEXT, BUT1, "Back", ERR)

      * change the button responses
      CALL ChangeNVContacts(CONTEXT, BUT1GRPID, BUT1, 2, BUT1GRP.RESP, ERR)

      * clear the flag to show that the buttons have been swapped back
      SWITCH = 0
   END
RETURN


***************************************************************
*
* SUBROUTINE: CHANGE.MENUBAR
```

```
*
* PURPOSE: To demonstrate how the commands on the application's
*          menu bar might be changed, according to the menu
*          displayed.
*
* COMMENTS:
*    If the main menu is displayed, button group 2 and the Options
*    menu are disabled. Otherwise they are enabled. The ID
*    variable is tested to determine which is required.
*    Note that in the case of the menu, both the menuitem
*    group and the menu itself must be enabled or disabled.
*
***************************************************************

CHANGE.MENUBAR:
   * if the main menu is displayed
   IF ID = "MAIN" THEN
      * disable button group 2
      CALL SetEnabledNVGroup(CONTEXT, BUT2GRPID, FALSE, ERR)

      * disable the Options menu
      CALL SetEnabledNVGroup(CONTEXT, MENU2GRPID, FALSE, ERR)
      CALL SetEnabled(CONTEXT, APPMENUOPTIONS, FALSE, ERR)

   * otherwise
   END ELSE
      * enable button group 2
      CALL SetEnabledNVGroup(CONTEXT, BUT2GRPID, TRUE, ERR)

      * enable the Options menu
      CALL SetEnabledNVGroup(CONTEXT, MENU2GRPID, TRUE, ERR)
      CALL SetEnabled(CONTEXT, APPMENUOPTIONS, TRUE, ERR)
   END
RETURN


***************************************************************
*
* SUBROUTINE: RUN.CALC
*
* PURPOSE: To run the Windows calculator.
*
* COMMENTS:
```

```
*    This routine illustrates how menu options can be used to run
*    utililities from within a NewView application. The utility
*    could just as easily be a DATA/BASIC application.
*
***************************************************************

RUN.CALC:
   COMMANDLINE = "calc.exe"           ;* file to execute
   WINDOWSTATE = EXECUTE.SHOWNORMAL   ;* window state
   CONTROL = EXECUTE.WAIT             ;* don't return until closed
   CALL Execute(COMMANDLINE, WINDOWSTATE, CONTROL, ERR)
   IF ERR THEN
      * if ERR is non-zero, there was an error
      ERRMSG = "Unable to run Calculator : ":ERR ;* error message
      GOSUB ERRSUB                ;* handle the error
   END
   * Set the focus back to the terminal window
   CALL SetContactFocus(CONTEXT, CHILDWIN, ERR)
RETURN


***************************************************************
END
```

Header File

The header file contains constant definitions which are common to both the DATA/BASIC source code and the resource script. These definitions must exist as an item in the REALITY file containing the source of the MENUNV application, and also be available on the PC when compiling the resource script. The RealLink LanFTU file transfer facility can be used to copy the information from the host to the PC or vice versa.

```
***************************************************************
*
* MENUNV.H - Constant definitions NewView version of MENUEX program
*
***************************************************************


* Ids for App window and major components

EQU APPWIN        TO 101
EQU CHILDWIN      TO 102
EQU APPMENUBAR    TO 103
```

```
* Menu and MenuItem Ids

EQU APPMENUEDIT    TO 104
EQU APPMENUFILE    TO 108
EQU APPMENUEXIT    TO 109
EQU APPMENUOPTIONS TO 149
EQU APPMENUDIARY   TO 150
EQU APPMENUCALC    TO 151
EQU APPMENUCHANGE  TO 152


* Button Ids

EQU BUT1          TO 201
EQU BUT2          TO 202
EQU BUT3          TO 203
EQU BUT4          TO 204
EQU BUT5          TO 205
EQU BUT6          TO 206
EQU BUT7          TO 207
EQU BUT8          TO 208
EQU BUT9          TO 209
EQU BUT10         TO 210


* Group ids

EQU HOTGRPID      TO 1
EQU BUT1GRPID     TO 2
EQU BUT2GRPID     TO 3
EQU MENU1GRPID    TO 4
EQU MENU2GRPID    TO 5


* Button sizes and positions

EQU BUTWIDTH      TO 100
EQU BUTHEIGHT     TO 50
EQU BUTY          TO 0
EQU BUT1X         TO 0
EQU BUT2X         TO 100
EQU BUT3X         TO 200
EQU BUT4X         TO 300
EQU BUT5X         TO 400
EQU BUT6X         TO 500
EQU BUT7X         TO 600
EQU BUT8X         TO 700
EQU BUT9X         TO 800
EQU BUT10X        TO 900


* horizontal position of TE window in App window; allows for button
bar

EQU TEWINSTART TO 55


* Additional menu responses

EQU SWAP.RESP    TO 'S'
EQU OK.RESP      TO ''
EQU CALC.RESP    TO 'C'
EQU DIARY.RESP   TO 'D'


* general constants

EQU CRET TO CHAR(13)
EQU AM TO CHAR(254)
```

Resource Script

The resource script must be created on the PC and given a name with the extension '.UCL'. It contains the definitions for the various UIMS contacts used in MENUNV.

```
***************************************************************
*
* MENUNV.UCL - Resource file for NewView version of MENUEX program
*
***************************************************************

#INCLUDE RFWDEFS.H
#INCLUDE MENUNV.H

* Main application window and menus
APPWINDOW = APPWIN
{
   TITLE = 'NewView Demonstration'
   POSITION = 0, 40
   SIZE = 1000, 800
   STYLE = NOSCROLL, MOVABLE, ICONISABLE
   BDRSTYLE = BORDER

   MENUBAR = APPMENUBAR
   {
      * The File menu has all the RealLink Printing commands plus
Exit
      MENU = APPMENUFILE
      {
         TITLE = '&File'
         ENABLED = TRUE
         CHILDREN = '&Print'          = ID.FILEPRINT,
                    'Print &Window'   = ID.FILEPRINTWINDOW,
                    'P&rinter Setup' = ID.FILEPRINTERSETUP,
                    '-'               = 0,
                    'E&xit'           = APPMENUEXIT
      }

      * The Edit menu has the three RealLink editing commands
      MENU = APPMENUEDIT
      {
         TITLE = '&Edit'
         ENABLED = TRUE
         CHILDREN = '&Copy'           = ID.EDITCOPY,
                    '&Paste'          = ID.EDITPASTE,
                    '-'               = 0,
                    'Copy &Window'    = ID.EDITCOPYWINDOW
      }

      * The Options menu has Diary, Calculator and Swap commands
      MENU = APPMENUOPTIONS
      {
         TITLE = '&Options'
         ENABLED = FALSE
         CHILDREN = '&Diary'          = APPMENUDIARY,
                    '&Calculator'     = APPMENUCALC,
                    '&Swap strings'   = APPMENUCHANGE
      }
   }
```

```
* Child window to use as the terminal window
CHILDWINDOW = CHILDWIN
{
   POSITION = 0, TEWINSTART
   SIZE = 1000, 730
   STYLE = NOSCROLL
   BDRSTYLE = NONE
   EVENTMASK = NEWVIEW
}

* Buttons

TITLEDBUTTON = BUT1
{
   TITLE = 'Back'
   POSITION = BUT1X, BUTY
   MAPPED = FALSE
   ENABLED = TRUE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT2
{
   TITLE = 'Main'
   POSITION = BUT2X, BUTY
   MAPPED = FALSE
   ENABLED = TRUE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT3
{
   TITLE = ''
   POSITION = BUT3X, BUTY
   MAPPED = FALSE
   ENABLED = FALSE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT4
{
   TITLE = ''
   POSITION = BUT4X, BUTY
   MAPPED = FALSE
   ENABLED = FALSE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT5
{
   TITLE = ''
   POSITION = BUT5X, BUTY
   MAPPED = FALSE
   ENABLED = FALSE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT6
{
   TITLE = ''
   POSITION = BUT6X, BUTY
   MAPPED = FALSE
   ENABLED = FALSE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT7
{
   TITLE = ''
   POSITION = BUT7X, BUTY
   MAPPED = FALSE
   ENABLED = FALSE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT8
{
   TITLE = ''
   POSITION = BUT8X, BUTY
   MAPPED = FALSE
   ENABLED = FALSE
   SIZE = BUTWIDTH, BUTHEIGHT
}

TITLEDBUTTON = BUT9
{
      TITLE = 'Ok'
      POSITION = BUT9X, BUTY
      MAPPED = FALSE
      ENABLED = FALSE
      SIZE = BUTWIDTH, BUTHEIGHT
   }

   TITLEDBUTTON = BUT10
   {
      TITLE = 'Swap'
      POSITION = BUT10X, BUTY
      MAPPED = FALSE
      ENABLED = FALSE
      SIZE = BUTWIDTH, BUTHEIGHT
   }
}
```

Menu Definition File

MENUNV uses the same menu definition file as MENUEX. Refer to page A-6 for details.

Active Window	The window which the user can currently manipulate or work with. This is similar to having the **focus**.
API	Application Programming Interface.
App Window	An App window is the main type of window in a UIMS application. It is free to appear anywhere on the screen and to overlap any other window (compare **Child Window**). A UIMS application must have at least one App window (the root window).
Attribute	1. A unique characteristic of an **object** that can be modified. 2. A section of a REALITY file item, delimited by attribute marks – CHAR(254).
Brush	1. The way the interior of a graphical object looks; it can be coloured, hatched, or patterned. 2. A UIMS **object** that controls these characteristics.
Check Button	A check button is a **control** that can be turned on or off and saves its state. It looks like a square box to the left of some text. If it has been selected, an 'X' appears in the box.
Check Mark	A mark shown beside a menu item to indicate a selected option. The mark displayed is normally a tick (✓), but on some hardware platforms other marks may be used.
Child Window	A child window is similar to an **App window**, but cannot overlap windows other than its parent.
Client Area	The client area is the part of a window where an application can draw. It is usually the central area of the window and excludes the title area, menu bar, scroll bars, etc.
Client Coordinates	Coordinates relative to the top left-hand corner of the window's **client area**.
Clip Region	Defines in which part of a window drawing can take place. An application may draw outside the clip region, but only the part inside the clip region will be displayed.
Clipboard	The clipboard can be thought of as a resting place in memory for data that has been copied or cut from one application to be pasted into the same or a different application.
Contact	An **object** that provides an interface with the user. Window, menu, and dialogue box objects are all contacts.

Context	An **object** that defines certain application wide parameters, such as the coordinate mode, the default drawing objects, and the event mask.
Control	A control is a **contact** that carries out a specific kind of input or output. Edit boxes, titled buttons and scroll bars are examples of controls.
Copy	To Copy means to get data from an application and put it in the **clipboard**.
Cursor	A blinking graphic entity that shows where the next text input will appear on the screen.
Cut	To Cut means to get some data from an application and put it in the **clipboard** and then to remove the data from the application.
DDE	Dynamic Data Exchange – a message exchange protocol used in the Microsoft Windows environment.
Default Titled Button	A default **titled button** is a control that represents the usual response to a request. It has text surrounded by an emboldened rectangle. If the user types the RETURN key it is the default titled button that takes effect.
Dialog Box	A dialog box is a window that an application displays to request information from the user. It contains **controls** that the user can manipulate.
Disabled	If an application does not want to allow the user to select a particular option at a certain time, it can disable the option. Disabling a contact causes any text in the contact to be grayed.
Edit Control	A **control** that lets the user type in his own text.
Enabled	Selectable by the user.
Event	Actions carried out by the user result in UIMS events, the details of which are sent to the application by means of **messages**. For example, when the user presses a key, the resulting event generates a keypress message, which tells the application which key was pressed.
Focus	If a window has the focus, all keyboard events will be sent to that window.
Font	The typeface used to display text.
GUI	Graphical User Interface.
Instance	An occurrence of an application.

List Box	A list box is a **control** that presents the user with a list of options which may be clicked on to accomplish some action. Often there is a **scroll bar** attached to the list box to scroll through the options, which may be numerous. A common use of a list box is to present the user with a list of files to select from.
Menu	A menu is a list of action choices listed at the top of a window that can be selected with a pointing device or from the keyboard.
Message	UIMS communicates with applications by passing predefined formatted messages. Examples are messages which tell the application to paint its window, and messages which tell the application that the user has selected a command on the menu.
Object	A software packet containing a collection of related data (in the form of attributes) and procedures for operating on that data.
Option Button	An option button is a **control** that usually appears in a group of other option buttons. Each choice is mutually exclusive of the others in the group, so that once the user selects one button, any other button in the group turns off. Selecting a option button is analogous to selecting a radio station on a car radio; for this reason, option buttons are often called radio buttons.
Parent	An **object** or **contact** to which other objects or contacts are attached. For example, a dialog box is the parent of the controls it contains.
Paste	A command to insert the current contents of the **clipboard** into an application's window.
Pen	The way the outline of a graphical **object** looks. It can be wide, coloured, or patterned.
Pointer	A graphic entity that is controlled by a **pointing device** to make selections in an application's window.
Pointing Device	A pointing device is an input device used to control the **pointer** on the screen. It can be a mouse, a light pen, a joystick or a graphic tablet.
Resource Compiler	The resource compiler converts a text file that describes the resources (menus, dialog boxes, etc.) used by an application into the format required by the application.
Screen Coordinates	Coordinates relative to the top left corner of the display.
Scroll Bar	A scroll bar is a **control** that allows the user to set analogue values. Its main use is to let the user change the current view of the application when there is more data than can be displayed in one window.

System Menu	The system menu is a special menu that is pulled down from the top left corner of a window. It contains actions that are usually common to all applications such as moving or changing the size of the window.
Thumb	A part of a **scroll bar** that can be dragged with the mouse to change the scroll bar setting. Its position on the scroll bar indicates the current setting.
Title Bar	The title bar is the uppermost part of a window that provides two pieces of information; the name of the application and whether the window is currently active. Another name for the title bar is the caption bar.
Titled Button	A titled button is a **control** that has text surrounded by a rectangle. Clicking on it causes an immediate reaction. For example, in dialog boxes there are OK and Cancel buttons. Titled buttons are also known as Push Buttons.